Liceo Compu-Market

Catedrático: Erick González

Cátedra: Programación

Tema:

Temario Libre de Programación

Nombre: SheiferGabriela

Iguardia Lopez

Grado: 5to.

Bachillerato en computación

Fecha:

20 de abril de 2017

**INDICE**

Índice………………………………………………………………………………………………... 2

Introducción………………………………………………………………………………………… 4

Programación……………………………………………………………………………….….. 5

Lenguajes de programación………………………………………………………….….. 9

Fase de diseño de programas y puesta a punto……………………………….. 11

Programa………………………………………………………………………………………… 13

Tipos de Programas…………………………………………………………………………. 14

Estructura general de un lenguaje en pascal……………………………………. 17

Paradigmas de la programación……………………………………………………….. 18

Diseño de programas estructurados…………………………………………………. 25

Programación modular……………………………………………………………………… 26

Subprogramas…………………………………………………………………………………… 29

Cohesión y acoplamiento………………………………………………………………….. 32

Documentación de programas………………………………………………………….. 36

Directrices de la documentación……………………………………………………….. 43

Programación orientada a objetos…………………………………………………….. 46

Características del lenguajes C………………………………………………………….. 48

Operadores aritméticos y de asignación……………………………………………. 56

Salida, Entrada………………………………………………………………………………….. 60

Sentencia Scanf…………………………………………………………………………………. 62

Operadores lógicos……………………………………………………………………………. 68

Bucles………………………………………………………………………………………………… 69

Sentencia Do…While………………………………………………………………………….. 70

Sentencia for……………………………………………………………………………………… 71

Sentencia break, continue………………………………………………………………….. 72

Funciones………………………………………………………………………………………..… 74

Arrays………………………………………………………………………………………………… 80

Punteros……………………………………………………………………………………………. 85

Estructuras……………………………………………………………………………………….. 86

Typedef…………………………………………………………………………………………….. 93

Ficheros……………………………………………………………………………………………. 94

Gestion dinámica de memoria…………………………………………………………. 102

Progrmacion grafica…………………………………………………………………….….. 120

Apendice…………………………………………………………………………………..……… 125

JavaScript………………………………………………………………………………..………. 130

Recursos sobre JavaScript………………………………………………………………… 132

JQuery…………………………………………………………………………………………….. 134

Mootools…………………………………………………………………………………..……. 135

Bootstrap………………………………………………………………………………………… 136

SQL……………………………………………………………………………….…………………. 137

Visual Basic………………………………………………………………….………………….. 138

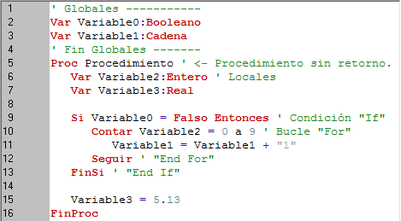
Conclusiones……………………………………………………………………..…………….. 139

**INTRODUCCIÓN**

**PROGRAMACIÓN:**

Es el proceso de planificar un conjunto de actividades o tareas que se van a realizar a partir que se produce el planteo del problema hasta obtener una solución instalada en la computadora. El concepto de programa almacenado en memoria fue introducido por Jon Von Newman en 1946 lo que hizo más flexibles a los programas de computadora.

## HISTORIA:

[](https://commons.wikimedia.org/wiki/File:Pauscal_lenguaje_de_programaci%C3%B3n.png)

Para crear un [programa](https://es.wikipedia.org/wiki/Programa_inform%C3%A1tico), y que la computadora lo interprete y ejecute las instrucciones escritas en él, debe escribirse en un [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n). En sus inicios las computadoras interpretaban solo instrucciones en un lenguaje específico, del más bajo nivel, conocido como [código máquina](https://es.wikipedia.org/wiki/C%C3%B3digo_m%C3%A1quina), siendo éste excesivamente complicado para programar. De hecho solo consiste en cadenas de números 1 y 0 ([sistema binario](https://es.wikipedia.org/wiki/Sistema_binario)). Para facilitar el trabajo de programación, los primeros científicos, que trabajaban en el área, decidieron reemplazar las instrucciones, secuencias de unos y ceros, por palabras o abreviaturas provenientes del [inglés](https://es.wikipedia.org/wiki/Idioma_ingl%C3%A9s); las codificaron y crearon así un lenguaje de mayor nivel, que se conoce como Assembly o [lenguaje ensamblador](https://es.wikipedia.org/wiki/Lenguaje_ensamblador). Por ejemplo, para sumar se podría usar la letra A de la palabra inglesa *add* (sumar). En realidad escribir en lenguaje ensamblador es básicamente lo mismo que hacerlo en lenguaje máquina, pero las letras y palabras son bastante más fáciles de recordar y entender que secuencias de números binarios. A medida que la complejidad de las tareas que realizaban las computadoras aumentaba, se hizo necesario disponer de un método sencillo para programar. Entonces, se crearon los [lenguajes de alto nivel](https://es.wikipedia.org/wiki/Lenguaje_de_alto_nivel). Mientras que una tarea tan trivial como multiplicar dos números puede necesitar un conjunto de instrucciones en lenguaje ensamblador, en un lenguaje de alto nivel bastará con solo una. Una vez que se termina de escribir un programa, sea en ensamblador o en algunos lenguajes de alto nivel, es necesario compilarlo, es decir, traducirlo completo a lenguaje máquina.[1](https://es.wikipedia.org/wiki/Programaci%C3%B3n) Eventualmente será necesaria otra fase denominada comúnmente [*link* o enlace](https://es.wikipedia.org/wiki/Enlazador), durante la cual se anexan al código, generado durante la compilación, los recursos necesarios de alguna [biblioteca](https://es.wikipedia.org/wiki/Biblioteca_(inform%C3%A1tica)). En algunos lenguajes de programación, puede no ser requerido el proceso de compilación y enlace, ya que pueden trabajar en modo [intérprete](https://es.wikipedia.org/wiki/Int%C3%A9rprete_(inform%C3%A1tica)). Esta modalidad de trabajo es equivalente pero se realiza [instrucción](https://es.wikipedia.org/wiki/Instrucci%C3%B3n_inform%C3%A1tica) por instrucción, a medida que es ejecutado el programa.

## OBJETIVOS DE LA PROGRAMACIÓN

La programación debe perseguir la obtención de programas de calidad. Para ello se establece una serie de factores que determinan la calidad de un programa. Algunos de los factores de calidad más importantes son los siguientes:

* [**Correctitud**](https://es.wikipedia.org/wiki/Correctitud)**.** Un programa es correcto si hace lo que debe hacer tal y como se estableció en las fases previas a su desarrollo. Para determinar si un programa hace lo que debe, es muy importante especificar claramente qué debe hacer el programa antes de su desarrollo y, una vez acabado, compararlo con lo que realmente hace.
* **Claridad.** Es muy importante que el programa sea lo más claro y legible posible, para facilitar tanto su desarrollo como su posterior mantenimiento. Al elaborar un programa se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo de programación. De esta forma se ve facilitado el trabajo del [programador](https://es.wikipedia.org/wiki/Programador), tanto en la fase de creación como en las fases posteriores de corrección de errores, ampliaciones, modificaciones, etc. Fases que pueden ser realizadas incluso por otro programador, con lo cual la claridad es aún más necesaria para que otros puedan continuar el trabajo fácilmente. Algunos programadores llegan incluso a utilizar [Arte ASCII](https://es.wikipedia.org/wiki/Arte_ASCII) para delimitar secciones de código; una práctica común es realizar aclaraciones en el código fuente utilizando *líneas de comentarios*. Contrariamente, algunos por diversión o para impedirle un análisis cómodo a otros programadores, recurren al uso de [código ofuscado](https://es.wikipedia.org/wiki/C%C3%B3digo_ofuscado).
* **Eficiencia.** Se trata de que el programa, además de realizar aquello para lo que fue creado (es decir, que sea correcto), lo haga gestionando de la mejor forma posible los recursos que utiliza. Normalmente, al hablar de eficiencia de un programa, se suele hacer referencia al tiempo que tarda en realizar la tarea para la que ha sido creado y a la cantidad de memoria que necesita, pero hay otros recursos que también pueden ser de consideración para mejorar la [eficiencia](https://es.wikipedia.org/wiki/Eficiencia) de un programa, dependiendo de su naturaleza (espacio en disco que utiliza, tráfico en la red que genera, etc.).
* [**Portabilidad**](https://es.wikipedia.org/wiki/Portabilidad)**.** Un programa es portable cuando tiene la capacidad de poder ejecutarse en una plataforma, ya sea [hardware](https://es.wikipedia.org/wiki/Hardware) o [software](https://es.wikipedia.org/wiki/Software), diferente a aquella en la que se desarrolló. La portabilidad es una característica muy deseable para un programa, ya que permite, por ejemplo, a un programa que se ha elaborado para el sistema [GNU/Linux](https://es.wikipedia.org/wiki/GNU/Linux) ejecutarse también en la familia de sistemas operativos [Windows](https://es.wikipedia.org/wiki/Windows). Esto permite que el programa pueda llegar a más usuarios más fácilmente.

## Ciclo de vida del software

## El término [ciclo de vida del software](https://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software) describe el desarrollo de software, desde la fase inicial hasta la fase final, incluyendo su estado funcional. El propósito es definir las distintas fases intermedias que se requieren para validar el desarrollo de la aplicación, es decir, para garantizar que el software cumpla los requisitos para la aplicación y verificación de los procedimientos de desarrollo: se asegura que los métodos utilizados son apropiados. Estos métodos se originan en el hecho de que es muy costoso rectificar los errores que se detectan tarde dentro de la fase de implementación (programación propiamente dicha), o peor aun, durante la fase funcional. El modelo de ciclo de vida permite que los errores se detecten lo antes posible y por lo tanto, permite a los desarrolladores concentrarse en la calidad del software, en los plazos de implementación y en los costos asociados. El ciclo de vida básico de un software consta de, al menos, los siguientes procedimientos:

* Definición de objetivos: definir el resultado del proyecto y su papel en la estrategia global.
* Análisis de los requisitos y su viabilidad: recopilar, examinar y formular los requisitos del cliente y examinar cualquier restricción que se pueda aplicar.
* Diseño general: requisitos generales de la arquitectura de la aplicación.
* Diseño en detalle: definición precisa de cada subconjunto de la aplicación.
* Programación (programación e implementación): es la implementación en un lenguaje de programación para crear las funciones definidas durante la etapa de diseño.
* Prueba de unidad: prueba individual de cada subconjunto de la aplicación para garantizar que se implementaron de acuerdo con las especificaciones.
* Integración: para garantizar que los diferentes [módulos](https://es.wikipedia.org/wiki/M%C3%B3dulos) y [subprogramas](https://es.wikipedia.org/w/index.php?title=Subprogramas&action=edit&redlink=1) se integren con la aplicación. Este es el propósito de la prueba de integración que debe estar cuidadosamente documentada.
* Prueba beta (o validación), para garantizar que el software cumple con las especificaciones originales.
* Documentación: se documenta con toda la información necesaria, sea funcional final para los usuarios del software (manual del usuario), y de desarrollo para futuras adaptaciones, ampliaciones y correcciones.
* Mantenimiento: para todos los procedimientos correctivos (mantenimiento correctivo) y las actualizaciones secundarias del software (mantenimiento continuo).

El orden y la presencia de cada uno de estos procedimientos en el ciclo de vida de una aplicación dependen del tipo de modelo de ciclo de vida acordado entre el cliente y el equipo de desarrolladores. En el caso del [software libre](https://es.wikipedia.org/wiki/Software_libre) se tiene un ciclo de vida mucho más dinámico, puesto que muchos programadores trabajan en simultáneo desarrollando sus aportaciones.

**LENGUAJES DE PROGRAMACIÓN:**

|  |
| --- |
| Los lenguajes de programación son herramientas que nos permiten crear programas y software. Entre ellos tenemos Delphi, Visual Basic, Pascal, Java, etc..  Una computadora funciona bajo control de un programa el cual debe estar almacenado en la unidad de memoria; tales como el disco duro.  Los *lenguajes de programación* de una computadora en particular se conoce como código de máquinas o lenguaje de máquinas. |

Estos lenguajes codificados en una computadora específica no podrán ser ejecutados en otra computadora diferente.  
  
Para que estos programas funcionen para diferentes computadoras hay que realizar una versión para cada una de ellas, lo que implica el aumento del costo de desarrollo.  
  
Por otra parte, los lenguajes de programación en código de máquina son verdaderamente difíciles de entender para una persona, ya que están compuestos de códigos numéricos sin sentido nemotécnico.  
  
Los *lenguajes de programación* facilitan la tarea de programación, ya que disponen de formas adecuadas que permiten ser leidas y escritas por personas, a su vez resultan independientes del modelo de computador a utilizar.  
  
Los lenguajes de programación representan en forma simbólica y en manera de un texto los códigos que podrán ser leidos por una persona.   
  
Los lenguajes de programación son independientes de las computadoras a utilizar.

Existen estrategias que permiten ejecutar en una computadora un programa realizado en un lenguaje de programación simbólico. Los procesadores del lenguaje son los programas que permiten el tratamiento de la información en forma de texto, representada en los lenguajes de programación simbólicos.  
  
Hay lenguajes de programación que utilizan *compilador.*  
  
  
La ejecución de un programa con compilador requiere de dos etapas:  
  
1) Traducir el programa simbólico a código máquina  
2) Ejecución y procesamiento de los datos.  
  
Otros lenguajes de programación utilizan un programa intérprete o traductor, el cual analiza directamente la descripción simbólica del programa fuente y realiza las instrucciones dadas.  
  
El *intérprete* en los lenguajes de programación simula una máquina virtual, donde el lenguaje de máquina es similar al lenguaje fuente.  
  
La ventaja del proceso interprete es que no necesita de dos fases para ejecutar el programa, sin embargo su inconveniente es que la velocidad de ejecución es más lenta ya que debe analizar e interpretar las instrucciones contenidas en el programa fuente.

**FASES DE DISEÑO DE PROGRAMAS Y PUESTA A PUNTO:**

**Fases de Diseño:**

**Análisis:**Es el momento en que se analiza la situación y se determina cual es el problema a resolver. También es el momento de analizar cuales serán los datos de entrada y que datos se desean obtener como salida como así también los vínculos que los relacionan sin darle demasiada importancia al proceso de transformación de los datos, es decir se lo ve como una caja negra donde entra información y salen los resultados.

**Algoritmo:**Es el proceso de considerar y desarrollar una secuencia lógica de pasos para resolver el problema.

**Prueba de Algoritmo:** Es el seguimiento de la secuencia de pasos lógicos del algoritmo para ver si realmente resuelven el problema.

**Codificación:** Es la conversión del algoritmo en un programa a través de un lenguaje de programación especifico.

**Edición, Ejecución y Prueba:** Es el proceso de introducir o cargar el programa en memoria, ejecutarlo y probar los resultados que arroja corrigiendo hasta la puesta a punto final del mismo.

**Uso y Mantenimiento:** Es el periodo de uso del programa durante su vida útil. En este periodo con seguridad surgirán con el uso nuevos errores no advertidos en etapas anteriores los cuales deberán ser corregidos, como así también se le realizaran cambios estructurales si es necesario como ser agregarle nuevas funciones o cambiar la estructura del mismo según los requerimientos del usuario con el fin de maximizar la optimización del mismo hasta el final de su vida útil.

**PUESTA A PUNTO:**

**Concepto:** Es el conjunto de tareas necesarias para implementar el programa en la computadora. Comprende las siguientes fases:

**Fase de Edición:** Es el proceso de escribir el programa con un editor de texto para que quede almacenado en la memoria de la PC. A este se lo llama “Código Fuente”.

**Fase de Traducción:** Se traduce el programa que esta en un lenguaje de alto nivel a un lenguaje de bajo nivel entendible para la computadora. De esta manera se obtiene el programa objeto o ejecutable.

**Fase de Montaje:** Consiste en enlazar objetos previamente compilados, es decir el uso de librerías.

**Fase de ejecución:** Uso del programa ya cargado en memoria para verificar si sus resultados son correctos.

**PROGRAMA**

Es un conjunto ordenado de sentencias o instrucciones que le dicen a la computadora las tareas o acciones a realizar para resolver un problema planteado de antemano.

**Partes constitutivas de un programa**:

**Entrada de Datos:** Es el proceso de introducir información en la memoria de la computadora. Para ello se debe realizar una lista de todas las entradas requeridas por el programa coma si también la fuente y formato de procedencia y la frecuencia con que serán requeridas cada una de ellas. El programador deberá prevenir situaciones en las que un dato o un conjunto de datos sean requeridos por el programa sin antes haber sido introducido en memoria.

**Algoritmos de Resolución / Codificación:** Consta de dos pasos:

**Diseño del modelo de resolución del problema**: Se establece el modelo proceso para la resolución del problema para lo cual se tendrá en cuenta los datos de entrada y los resultados que se desean obtener.

**Algoritmo de Resolución del Problema:** Suele expresarse previamente a la codificación en el lenguaje de programación. Estos pueden ser:

* Pseudocódigo.
* Diagrama de Flujo.
* Diagrama Estructurado o N - S (Nassi \_ Shneiderman).

**Salida de Resultados:** Es el proceso de la obtención de los datos ya transformados. Estos deben ser previstos por el programador tanto en tipo y formato como así también en los tiempos de salida de los mismos. Estos pueden ser:

* Salida por pantalla o en papel impreso.
* Grabación en disco o cinta.
* Diseño de Presentación (tablas, gráficos, listas, informes).
* Frecuencia de Salida.

**TIPOS DE PROGRAMAS:**

**Lineales o Secuenciales:** En este tipo de programas las instrucciones se ejecutan una a continuación de la otra, es decir en una secuencia preestablecida sin la posibilidad de saltos condicionales o incondicionales.

**Cíclicos:** En estos tipos de programas un conjunto de instrucciones se ejecuta (repite) un determinado número de veces hasta que se cumpla una condición.

**Alternativos:** En estos tipos de programas se ejecuta un determinado grupo de instrucciones entre n grupo de instrucciones distintas entre si. La ejecución de uno de ellos resulta de la evaluación de una condición o del estado de alguna variable.

Para n = 2

**ELEMENTOS BÁSICOS DE UN PROGRAMA:**

Son aquellos que combinados adecuadamente nos permiten construir un programa. Estos son:

**Palabras Reservadas:**Constituyen las instrucciones intrínsecas al lenguaje de programación y son la parte fundamental de su sintaxis. No pueden ser elegidos como identificadores o nombres de variables.

**Constantes:** Representan a un elemento fijo de dato, es decir que su valor no puede cambiar durante la ejecución de un programa.

**Variables**: Representan a elementos variables de datos, es decir que su contenido puede cambiar durante la ejecución de un programa.

**Expresiones:** Son combinaciones de constantes, variables, símbolos de operación, operadores, y su valor es el resultante de la resolución de dichas operaciones. Estas pueden ser:

Según su tipo:

* **Simples:** Están formadas por un elemento que va precedido por un operador lunario o el operador lógico not Ej.: -A, Not (B).
* **Compuestas:** Están formadas por dos o más elementos combinados con uno o más operadores Ej.: A + B, B + sin (C) \* 2.

Según el resultado que se obtiene:

**Aritméticas: Ej. 5 + 8 13**

**Lógicas: Pueden ser:**

**Relacionales: Ej. A > B V**

**Lógicas: Eje A And B F**

**Sentencias de Asignación:** Sirven para asignar el valor de una expresión a una variable.

**Elementos Auxiliares de un Programa**:

**Contadores:** Un contador es un campo de memoria que está destinado a contener los diferentes valores que se van incrementado o decrementando en cada iteración. Es decir que el campo controla las sucesivas sumas paralelas que se van realizando en la ejecución de las sucesivas repeticiones = C + N Donde N es una cantidad Constante.

**Acumuladores**: Un Acumulador o totalizador es un campo de memoria cuya misión es almacenar cantidades variables resultantes de sumas sucesivas. S = S + N Donde N es una cantidad Variable.

**Interruptores o Conmutadores:** A veces llamados centinelas, banderas (flags), es un campo de memoria (variable) que toman diversos valores a lo largo de la ejecución del programa y permite comunicar información de una parte a otra del mismo, es decir varias la secuencia de ejecución de un programa, dependiente del valor que tenga en un determinado momento. Los dos únicos valores que pueden tomar un interruptor son: 1 o 0, V o F, Encendido o Apagado.

Tipos de Instrucciones:

**Entrada /Salida:** Permiten la transferencia de información desde los periféricos de entrada a la memoria de la computadora y viceversa. El proceso de introducción de datos en la memoria se denomina lectura o carga de datos y el proceso de extracción de los mismos de la memoria a un dispositivo periférico de salida se denomina escritura, grabación o conservación, impresión, etc.

**Asignación:** Permiten asignar a una variable (la que se encuentra a la izquierda del operador de asignación) el valor de una expresión (que se encuentra a la derecha del operador de asignación).

**Movimiento:** Las instrucciones de movimiento tienen el mismo sentido que las de asignación y permiten transferir la información o contenido de un campo o posición de memoria a otro, manteniendo siempre intacta la información en su primera posición ( emisor) y variando el contenido de la segunda posición (receptor).

**Aritméticas:** Realizan el cálculo de operaciones aritméticas tales como sumas, restar, multiplicar, dividir o funciones como seno, coseno, tangente, etc.

Lógicas y de Relación: Este tipo de operaciones se realizan con operadores de relación (>, <,= <>, etc.) y operadores lógicos o Boléanos (and, or, not, etc.).

**Bifurcación o Transferencia de Control:** Permiten variar (romper) la secuencia de ejecución de un programa al saltar (bifurcar) a otra parte del mismo. Multiplicar, dividir o funciones como senos bifurcaciones pueden ser con salto adelante en la secuencia normal del programa o salto hacia atrás. En cuanto a las condiciones a cumplir son incondicionales y condicionales. Las incondicionales se realizan siempre que se ejecute la instrucción, mientras que las condicionales solo se realizan al cumplirse una determinada tarea.

**Especiales:** Estas instrucciones varían según cada lenguaje pero en general las mas importantes son: edición, impresión, conversión, ordenación, declarativas, modificación de direcciones, comunicación de Entrada/Salida, Graficas

.

**Estructura General de un Lenguaje en Pascal:**

|  |
| --- |
| Program identificador { cabecera del programa }  Uses identificadores  Label lista de etiquetas { sección de etiquetas }  const.  Definiciones de constantes  Type  Declaración de tipos de datos definidos por el usuario  Var.  Declaración de variables globales |
| Procedure  Definiciones de procedimientos  Function  Definición de funciones |
| Begin { cuerpo del programa }  Sentencias  End. |

**PARADIGMAS DE LA PROGRAMACIÓN:**

Los lenguajes y métodos de diseño son la guía para producir el diseño de un sistema. Un método de diseño es la forma de producir el diseño de un sistema. Los métodos más usados son: “Estructurado”, Diseño Descendente o “Top Down “y Diseño “Orientado a Objetos”. No obstante estas metodologías de diseño, dentro de los lenguajes de programación es común encontrar los llamados paradigmas de programación. Los lenguajes que soportan un paradigma especifico de computación se puede clasificar en orientado al paradigma de computación.

Los lenguajes de programación convencionales son una abstracción de la arquitectura de la computadora subyacente. El modelo abstracto consiste en la ejecución secuencia paso por paso, de las instrucciones que cambian el estado de la computación vía la modificación del almacén de valores. Los lenguajes convencionales basados en el modelo computacional Von Newman se llaman imperativos.

**PARADIGMAS DE PROGRAMACIÓN**

Un paradigma de programación es un modelo que engloba a ciertos lenguajes que comparten:

Elementos Estructurales ¿con qué se confeccionan los programas?

Elementos metodológicos: ¿cómo se confecciona un programa?

Consideramos los siguientes paradigmas

**Programación Imperativa:**Es la mas antigua (Maquina de Von Neumann).Un programa es una secuencia de acciones que se realizan en orden. Existen herramientas para modificar el orden de ejecución de las acciones.

**Programación Funcional:** definición de una serie de funciones.

**Programación Lógica:** definición de hechos y relaciones lógicas entre estos. No se indica el orden en el que se computa una función o se deriva un nuevo hecho.

**Programación Orientada a Objetos:** Un programa consiste en una colección de objetos que intercambian mensajes. Cada objeto es una entidad que agrupa una cierta información (estado) y un conjunto de mecanismos para manipularlas (métodos).

Caso Particular:

**Programación Orientada a Eventos:** Los lenguajes visuales orientada al evento y con manejo de componentes dan al usuario que no cuenta con mucha experiencia en desarrollo, la posibilidad de construir sus propias aplicaciones utilizando interfaces grafica sobre la base de ocurrencia de eventos. Para soportar este tipo de desarrollo interactúan dos tipos de herramientas, una que permite realizar diseños gráficos y un lenguaje de alto nivel que permite codificar los eventos. Con dichas herramientas es posible desarrollar cualquier tipo de aplicaciones basadas en el entorno.

Los paradigmas pueden ser considerados como patrones de pensamientos para la resolución de problemas, estos se clasifican en:

**Programación Estructurada:** En la programación estructurada se pueden conjuntas secuencias de declaraciones e instrucciones en un mismo lugar, las que después pueden ser usadas por el programa principal desde cualquier punto. La secuencia se conoce como subrutina, procedimiento o función dependiendo del lenguaje que se trate. Se dice que se hace una llamada para invocar al procedimiento. Después de que la secuencia es procesada, el control del flujo regresa al programa principal justo después de donde se hizo la llamada.

En resumen la programación estructurada consiste en dividir un programa en pequeñas piezas llamadas procedimientos y cada procedimiento es ejecutado o llamado por el programa principal desde diferentes lugares de este para resolver un problema.

El paradigma original de este tipo de programación es “Buscar los procedimientos que se desean; Usar los mejores algoritmos existentes”

**Programación Modular:** La programación modular permite agrupar procedimientos, que tienen una funcionalidad común, en módulos separados. Por lo tanto, un programa ya no consiste de una sola parte. El programa ahora se divide en varias partes más pequeñas que interactúan a través de interfaces y llamadas a procedimientos.

Cada módulo puede contener sus propios datos. Esto permite que cada módulo maneje un estado interno el cual es modificado por las llamadas a sus procedimientos. Sin embargo hay un único estado por modulo y cada módulo existe a lo más que una vez en todo programa. Típicamente requiero escribir dos archivos: la interfaz, donde se hacen todas las declaraciones de datos y funciones y la implementación. La interfaz describe lo que está disponible del módulo sin importar los detalles. De esta manera se esconde la información de la implementación. Este es un principio fundamental de la ingeniería de software.

El paradigma de la programación modular es: “Buscar los modulo que se requieran; particional al programa de tal manera que los datos se escondan dentro de los módulos”.

**Programación orientada a objetos:** Describen los lenguajes que soportan a objetos en interacción. Un objeto es un grupo de procedimientos que comparten un estado. El termino de orientado a objetos fue utilizado originalmente para distinguir aquellos lenguajes basados en objetos que soportaban clases de objetos y la herencia de atributos de un objeto padre por parte de sus hijos.

El paradigma de la programación orientada a objetos es: “Buscar las clases que se desean; proveer de un conjunto completo de operaciones para cada clase; especializar usando herencia”.

**Paradigmas declarativos:** Un lenguaje declarativo es uno en el que un programa especifica una relación o función. El intérprete o compilador para el lenguaje en particular administra la memoria por nosotros. Los tres paradigmas declarativos provienen de las matemáticas: la lógica, la teoría de funciones y el cálculo relacional.

**Paradigmas de programación lógica:** La programación lógica está basada en un subconjunto del cálculo de predicados, incluyendo instrucciones escritas en forma conocidas como cláusulas de Horn. Este paradigma puede deducir nuevos hechos a partir de otros hechos conocidos. Un sistema de cláusulas de Horn permite un método particularmente mecánico de demostración llamado resolución.

**Paradigmas funcionales:** Como su nombre lo dice operan solamente a través de funciones. Cada función devuelve un solo valor, dada una lista de parámetros. No se permiten asignaciones globales, llamados efectos colaterales. La programación funcional proporciona la capacidad para que un programa se modifique así mismo, es decir que pueda aprender.

**Paradigmas de la programación distribuida:** La programación concurrente ha sido dividida en dos amplias categorías, sistemas acoplados en forma débil o fuerte. El término distribuido se refiere por lo general a lenguajes para sistemas acoplados debidamente que soportan un grupo de programadores trabajando en un programa particular de manera simultánea y comunicándose a través de paso de mensajes mediante un canal de comunicación. Un sistema acoplado fuertemente permite que más de un proceso de ejecución tenga acceso a la misma ubicación de memoria. Un lenguaje acoplado con el sistema debe sincronizar el uso compartido de la memoria, de modo que solo un proceso escriba una variable compartida a la vez, y de modo que un proceso pueda esperar hasta que ciertas condiciones se satisfagan por completo antes de continuar la ejecución. La memoria compartida tiene la ventaja de la velocidad, porque no se necesita pasar mensajes.

**Paradigmas estructurados en bloques:** Se refiere a los ámbitos anidados, es decir los bloques pueden estar anidados dentro de otros bloques y pueden contener sus propias variables. La RAM representa una pila con una referencia al bloque que este actualmente activo en la parte superior.

**Paradigmas del lenguaje de base de datos:** Las propiedades que los distinguen de los lenguajes diseñados para tratar con bases de datos son la persistencia y la administración de cambios. Un sistema de administración de base de datos incluye un lenguaje de definición de datos (DDL) para describir una nueva colección de hechos, o datos y un lenguaje de manipulación de datos (DML) para la interacción con las bases de datos existentes.

**Programación Estructurada:**

Conceptos Básicos:

Los primitivos propósitos de la programación estructurada dirigían sus esfuerzos a buscar modos de minimizar la probabilidad de error en el proceso de programación. Uno de los objetivos de la programación estructurada es la minimización del error humano. Podríamos por ello enunciar la programación estructurada (PE) como una técnica de construcción de programas que utilizan al máximo los recursos del lenguaje, limita el conjunto de estructuras aplicables a leer y presenta una serie de reglas que coordinan adecuadamente el desarrollo de las diferentes fases de la programación.

La programación estructurada utiliza en su diseño los siguientes conceptos o principios fundamentales recogidos esencialmente en la definición anterior:

Estructuras básicas

Recursos abstractos

Diseño descendente “arriba-abajo” (Top-down)

Programa Propio:

Un Programa es considerado propio si cumple con las características siguientes:

Posee un solo punto de entrada y un solo punto de salida para facilitar el control del programa.

Si existen caminos que se pueden seguir desde la entrada hasta la salida y que pasen por todos los puntos del programa.

Si todas sus instrucciones son ejecutables y no existen lazos o bucles infinitos.

Tener un programa con una estructura clara se basa en tres principios fundamentales (Principios fundamentales de la Programación Estructurada).

Principios Fundamentales de la Programación Estructurada:

La Programación Estructurada se considera una técnica de programación sin saltos condicionales ni incondicionales. Un programa estructurado debe ser leído desde un principio hasta el fin de la secuencia normal de lectura y sin interrupciones. Los principios fundamentales de la Programación Estructurada son:

Estructuras básicas de Control:

**Secuencial:** En esta estructura una instrucción se ejecuta una a continuación de otra sin posibilidad de saltos condicionales o incondicionales.

**Alternativas o Selectivas:**

Permite elegir entre distintas alternativas o acciones a seguir dependiendo del valor que asuma la evaluación de una condición. El punto donde se toma o evalúa una decisión se denomina PUNTO DE DECISION y está representado por un rombo en un diagrama de flujo.

**Alternativa Simple:** (Condicional) Realiza la ejecución condicional de una acción, es decir que se evalúa la condición y si la condición se cumple, se ejecuta la acción.

Si {Condición} Entonces

{Acción}

Fin \_ si

**Alternativa Doble:** (Alternativa) Permite evaluar en tiempo de ejecución una condición boleaba para poder elegir entre dos alternativas posibles y distintas. Recordar que los valores resultantes de le evaluación de una condición son mutuamente excluyentes. Es verdadera o es falsa.

Si {Condición} Entonces

{Acción 1}

Si\_no

{Acción 2}

Fin \_ si

**Alternativa Múltiple:** (Selectiva) Permite asociar un conjunto de condiciones a un conjunto de acciones que se excluyen mutuamente, es decir que permite elegir entre “n” alternativas posibles y distintas

Según {Variable}

{Condición-1}: {Acción 1}

{Condición-2}: {Acción 2}

{Condición-3}: {Acción 3}

.

.

.

{Condición-n}: {Acción n}

Si\_no

{Acción x}

Fin \_ según

Estructuras Repetitivas:

Permite repetir la ejecución de una acción o un grupo de acciones un número determinado de veces, el que puede ser 0, 1 o más veces. En consecuencia un lazo o bucle es un conjunto de acciones que se van a repetir. Se especifica claramente que se debe repetir un número de veces, sin embargo ese número de veces a repetir puede ser conocido o no. Si es conocido significa que necesitamos información adicional y si no se conoce el número de veces, entonces está determinado por la evaluación de una condición.

**Estructura Para:** (For) Se emplea cuando por lo general el número de repeticiones es conocido de antemano y por lo general ese número no suele ser grande.

Para {Variable} de valor i hasta valor f incremento inc. hacer

{Acciones}

Fin \_ para

**Estructura Mientras:** (While) Esta Estructura determina la repetición de un grupo de instrucciones. Muestra la condición en forma inicial dado que se evalúa dicha condición antes de repetir la o las instrucciones asociadas. Es necesario que el valor de la evaluación de la condición cambia al menos una vez para evitar un lazo infinito.

Mientras {Condición} Hacer

{Acciones}

Fin \_ mientras

**Estructura Repetir:** (Repeat) Esta Estructura determina la repetición de un grupo de instrucciones. Muestra la condición al finalizar un conjunto de instrucciones donde esta es evaluada para determinar si se continúan repitiendo las instrucciones asociadas. Es necesario que dicha condición cambie al menos una vez para evitar un lazo o bucle infinito.

Repetir

{Acciones}

Hasta que {Condición}

Recursos Abstractos:

La estructuración debe cumplir el uso de recursos abstractos. El proceso de realización de diferentes pasos hasta encontrar la solución de un problema es un proceso abstracto.

Diseñar o concebir un problema en términos abstractos consiste en no tener en cuenta la máquina que lo va a resolver como así también el lenguaje de programación que se va a utilizar. Esto lleva consigo la obtención de un conjunto de acciones que se han de realizar para obtener una solución. Al considerar un algoritmo y los cálculos que incluyen se hace abstracción de los valores específicos.

**Diseño de Programas Estructurados:**

La realización del diseño estructurado de un programa se basa en la aplicación de los siguientes conceptos:

Ir de lo general a lo particular, descendiendo en la estructura del programa y en su nivel del detalle.

De la definición inicial del programa se pasa a un esquema del algoritmo descrito en pseudocódigo.

Independencia inicial del lenguaje.

Diseño por niveles, dejando los detalles para niveles posteriores. Verificar en cada nivel el esquema correcto.

Finalizar con un trabajo de pre composición del algoritmo completo.

Metodología Descendente “Arriba-Abajo” (Top-Down).

Consiste en establecer una serie de niveles de menor a mayor complejidad (arriba-abajo) que den solución al problema. En esencia consiste en efectuar una relación entre las etapas de la estructura de forma que una etapa jerárquica y su inmediatamente inferior se relacionen mediante entradas y salidas de información.

Un programa estructurado tiene una representación en forma de árbol.

El diseño se basa en la realización de diferentes niveles. El primer nivel resuelve totalmente el problema y el segundo y sucesivos niveles son “refinamientos sucesivos” del primero (stepwise) y se sigue siempre la metodología de recursos abstractos. Si el diseño y planteamiento es correcto nunca será preciso volver atrás ya que los niveles anteriores al que se esté situado en un momento dado ya habrán resuelto el problema en su totalidad.

**PROGRAMACIÓN MODULAR:**

El concepto básico de la programación modular es muy simple, consiste en dividir un programa en módulos. En realidad es un método de diseño que tiende a dividir el problema de forma lógica, en partes perfectamente diferenciadas que pueden ser analizadas programadas y puestas a punto independientemente. La división de un problema en módulos o programas independientes, exige otro modulo que controle y relacione a todos los demás, es el denominado modulo base o principal del problema.

Realmente la programación modular es un intento para diseñar programas de forma tal que cualquier función lógica pueda ser intercambiada sin afectar otras partes del programa.

Ventajas de la programación modular:

Un programa modular es más fácil de escribir y depurar (ejecutar, probar y poner a punto). Se puede profundizar en las pruebas parciales de cada módulo mucho más de lo que se hace un programa mayor.

Un programa modular es fácil de mantener y modificar.

Un programa modular es fácil de controlar. El desglose de un problema en módulos permite encomendar los módulos más complejos a los programadores más experimentados y los más sencillos a los programadores más noveles.

Posibilita el uso repetitivo de las rutinas en el mismo o en diferentes programas.

Inconvenientes de la programación modular:

No se dispone de algoritmos formales de modularidad, por lo que a veces los programadores no tienen claras las ideas de los módulos.

La programación modular requiere más memoria y tiempo de ejecución.

Objetivos de la programación modular:

* Disminuir la complejidad
* Aumentar la claridad y la fiabilidad
* Disminuir el coste
* Aumentar el control del proyecto
* Facilitar la ampliación del programa mediante nuevos módulos.
* Facilitar las modificaciones y correcciones al quedar automáticamente localizadas en un modulo

Un módulo está constituido por una o varias instrucciones físicamente contiguas y lógicamente encadenadas, las cuales se pueden referenciar mediante un nombre y pueden ser llamadas desde diferentes puntos de un programa.

Un módulo puede ser:

* Un programa
* Una función
* Una subrutina (o procedimiento).

Convocatoria de Módulos:

Se convoca un módulo con la llamada CALL, seguido con el nombre del Modulo

Esquema de Funcionamiento:

\*

\*

\*

\* \*

\*

\* Orden de Transferencia de Control.

Descripción:

Comienza con la ejecución de la primera instrucción del Programa principal.

Se continua ejecutando cada una de las instrucciones del módulo principal hasta que se encuentra una llamada a un subprograma.

Esta llamada implica una orden al procesador para que transfiera el control de la ejecución del programa a la primera instrucción del módulo llamado.

La ejecución continúa dentro del módulo llamado hasta que encuentra una orden de retorno o fin lógico. Luego se transfiera el control de ejecución del programa a la instrucción siguiente a la orden de llamada al subprograma.

Si un subprograma llama a su vez a otro subprograma se produce una inversión en el carácter del módulo dado que pasa a ser llamador y el otro el subprograma llamado. Como es de suponer, se vuelve a transferir el control del programa y se repite el proceso.

**SUBPROGRAMAS:**

Un medio de solucionar problemas es dividir el programa en partes que pueden ser desarrolladas por separado y eventualmente integrarse en una unidad. Tales partes independientes se llaman módulos o subprogramas.

Un subprograma puede ejecutar las mismas acciones que un programa, es decir aceptar datos, ejecutar cálculos y devolver o proporcionar resultados. Sin embargo un subprograma es utilizado por el programa para un fin específico. El subprograma recibe datos del programa y le devuelve resultados. Se dice que el programa llama o invoca al subprograma, este ejecuta una tarea y a continuación retorna o devuelve el control al programa desde donde fue hecha la llamada. Un subprograma puede tener también a su vez uno o varios subprogramas que se suelen conocer como subprogramas anidados o bien niveles jerárquicos...

El subprograma está constituido por un grupo de instrucciones que no forman parte del programa principal. Sin embargo, aunque cada subprograma es funcionalmente independiente (incluso ensamblado o compilado) no funciona nunca solo, debe siempre ser llamado por el programa principal y dejara de actuar cuando haya terminado su función y el control retorne al programa principal.

Existen dos tipos de subprogramas:

**Procedimientos o subrutinas:** Especifican cálculos que ejecutaran acciones en lugar de producir valores. Las subrutinas o procedimientos devuelven diferentes valores.

**Funciones:**especifican cálculos que producirán valores necesarios para evaluar expresiones. Las funciones devuelven un valor.

Parámetros de un Subprograma:

En general los subprogramas reciben información a través de parámetros formales

(O variables falsas). Cada invocación de un subprograma proporciona información por medio de parámetros actuales. Los parámetros actuales o efectivos especifican los valores actuales utilizados para parámetros formales en la definición de subprograma

Entrada Resultados

Tipos de Objetos:

Los diferentes objetos que manipulan un programa, ya sea constantes, variables, tablas, archivos, subprograma, etc., se clasifican según su ámbito de aplicación, es decir en que porción del programa y o subprogramas sus valores son conocidos y por lo tanto pueden ser utilizados.

Objetos Globales y Locales:

Son objetos globales los declarados en el programa principal, cuyo ámbito se extiende al mismo y a todos los subprogramas que lo componen.

Son objetos locales a un subprograma, los declarados en dicho subprograma y cuyo ámbito de aplicación, está restringido a ese subprograma y a todos los subprogramas en los declarados.

Pasó de Parámetros:

Como ya se dijo, el proceso de emisión y recepción de datos y resultados, mediante variables de enlace, se denomina paso de parámetros.

Un parámetro puede ser de dos tipos:

**Actuales o Reales:**Son los datos y variables enviados en cada llamada al subprograma por el programa o subprograma llamador. Ej.:

Nombre\_proc (pa1, pa2, pa3,….., pan);

**Fingidos, Formales o Ficticios:** Son las variables locales de un subprograma utilizada para la recepción y el envió de datos. Ej.:

Procedure Nombre\_proc (pf1, pf2, pf3,…..pfn);

El paso de parámetros puede realizarse de dos manera diferente:

**Paso por Valor:**Para suministrar datos de entrada al subprograma.

**Paso por Referencia:** Para entrada y Salida de datos o solo Salida

Un parámetros actual pasado por valor, es un dato o una variable global que contiene un valor y que es una entrada para el subprograma llamador. En el caso de ser una variable su contenido no puede ser modificado por el subprograma, dado que solamente se copia el contenido de la variable en el parámetro formal correspondiente cuyo valor luego será utilizado por el subprograma.

El parámetro actual pasado por referencia, es una variable del programa o subprograma llamador y que puede contener o no un valor.

El subprograma llameo coloca un resultado en esa variable el cual queda a disposición del programa llamador una vez concluida la ejecución del subprograma llamado.

Es de destacar que desde el punto de vista físico en el paso por valor no se proporciona la variable al subprograma, sino solamente su contenido evitando de esa manera su modificación.

En el paso por referencia se proporciona la dirección o referencia de la variable por lo cual el subprograma la utiliza como propia modificando su contenido si fuese necesario para dejar en ella los resultados que va a devolver. La utilización de parámetros por referencia, supone el ahorro de memoria puesto que la variable local o formal correspondiente no existe físicamente, sino que se asocia a la variable global correspondiente en cada llamada. Sin embargo supone el riesgo de modificar por error el contenido de una variable global sin desearlo.

**COHESIÓN Y ACOPLAMIENTO**:

**COHESIÓN:**

“La cohesión es la medida cualitativa de cuan estrechamente relacionados están los elementos internos de un módulo.”

**Cohesión Casual (la peor):**

La cohesión casual ocurre cuando existe poca o ninguna relación entre los elementos de un módulo. Es muy difícil encontrar módulos puramente casuales. Dificultan las modificaciones y mantenimiento de un programa.

**Cohesión Lógica (Sigue a la peor):**

Los elementos de un módulo están lógicamente asociados si puede pensarse en ellos como pertenecientes a la misma clase lógica de funciones, es decir aquellas que pueden pensarse como juntas lógicamente. Por ejemplo un módulo que agrupe elementos de procesamiento que caen en la clase de “entradas”. La cohesión lógica es más fuerte que la casual debido a que represente un mínimo de asociación entre el problema y los elementos del módulo pero un modelo lógicamente cohesivo no realiza una función específica sino que abarca una serie de funciones.

**Cohesión temporal (de moderada a pobre):**

Significa que todos los elementos de procesamiento de una colección ocurren en el mismo periodo de tiempo durante la ejecución del sistema .Existen conexiones entre la cohesión lógica y temporal pero la última es más fuerte ya que involucra al factor tiempo. Ejemplo un módulo que agrupe rutinas de inicialización de variables, apertura de archivos, etc.

**Cohesión de procedimiento (moderada):**

Se da cuando elementos de procesamiento relacionados procedural mente son elementos de una unidad procedural común. Una unidad procedural común puede ser un proceso de iteración (loop) y de decisión, o una secuencia lineal de pasos. Este nivel de cohesión comúnmente se tiene como resultado de derivar una estructura modular a partir de modelos de procedimiento como ser diagramas de flujo.

**Cohesión de comunicación (de moderada a buena):**

Es el menor nivel en el cual encontramos relación entre los elementos de procesamiento que es intrínsecamente dependiente del problema. Significa que todos los elementos operan sobre el mismo conjunto de datos de entrada o de salida. Ejemplos típicos pueden ser un módulo que imprima o grabe un archivo de transacciones o un modelo que reciba datos de diferentes fuentes y los transforme y ensamble en una línea de impresión.

**Cohesión secuencial:**

En ella los datos de salida (resultados) de un elemento de procesamiento sirven como datos de entrada al siguiente elemento de procesamiento. Este es claramente un principio asociativo relacionado con el dominio del problema.

**Cohesión funcional (la mejor):**

Es aquella que no es secuencial, por comunicación, por procedimiento, temporal, lógica o casual. En un módulo completamente funcional cada elemento de procesamiento es parte integral y esencial para la realización de una función simple. Por ejemplo un módulo que realiza una función matemática será puramente funcional.

“Cualquier modulo rara vez verifica un solo principio asociativo. Sus elementos pueden estar relacionados por una mezcla de los siete niveles de cohesión.

La cohesión de un módulo es aproximada al nivel más alto de cohesión que es aplicable a todos los elementos de procesamiento dentro del módulo”.

Acoplamiento:

“El acoplamiento es un concepto abstracto que nos indica el grado de interdependencia entre módulos”

Niveles de Acoplamientos y Tipos:

* (-) Bajo
* Datos
* Marcas o Estampados
* Control
* Externo
* Común
* Contenido
* (+)Alto

**Acoplamiento de Datos:**

Ocurre cuando dos o más módulos comparten el mismo elemento de datos no global.

Acoplamiento de Marcas o Estampados:

Ocurre cuando dos o más módulos comparten la misma estructura de datos no global.

Es decir que esos módulos comparten todo un conjunto de datos.

V es un Vector que es cargado

V

B es el menor valor del Vector V

**Acoplamiento de Control:**

Ocurre cuando un módulo le pasa a otro una bandera o interruptor booleano con el objetivo de controlar una lógica interna del módulo llamado. Es decir que la información transferida es únicamente dato de tipo bandera

B: Tipo Booleano

B

**Acoplamiento Externo:**

Se da cuando dos o más módulos se refieren al mismo elemento de datos globales

**Acoplamiento Común:**

Se da Cuando dos o más módulos se refieren a la misma estructura de datos Global.

**Acoplamiento de Contenido:**

Se da cuando un módulo puede transferir el control de ejecución a la mitad de otro modulo.

Criterios de modularizacion:

La división de un programa en módulos debe cumplir los siguientes criterios:

Cada módulo debe corresponder a una función lógica perfectamente diferenciada.

El tamaño de cada módulo es variable. Deben ser pequeños para que sean claros y de poca complejidad. Las normas varían según las situaciones. IBM considera que un módulo no debe superar el módulo de 50 instrucciones en PL/I. Como norma general práctica se puede considerar el tamaño máximo de un módulo como una página de listado de impresora.

Evitar variables externas.

Procurar no utilizar demasiados niveles de modularizacion para evitar complejidad de la red.

**DOCUMENTACIÓN DE PROGRAMAS:**

Un programa de computadora necesita siempre de una documentación cuya entidad será proporcional al volumen del mismo.

Existen tres grupos de personas que necesitan conocer la documentación del programa: programadores, operadores y usuarios. Los requerimientos necesarios para cada uno de ellos suelen ser diferentes en función de las misiones de cada grupo:

**Programadores:** Manual de mantenimiento del programa.

**Operadores:** Manual del operador. El operador es la persona encargada de correr el programa, introduciendo datos y extrayendo resultados.

**Usuarios:** Manual del usuario. El usuario es la persona de una organización que explota el programa, conociendo su función, las entradas requeridas, el proceso a ejecutar y la salida que produce.

En entornos interactivos como el caso de Turbo pascal, las misiones del usuario y operador suelen ser las mismas, así pues, la documentación del programa se pueden reducir a:

Manual del Usuario.

Manual de mantenimiento.

El manual de mantenimiento es la documentación requerida para mantener un programa durante su ciclo de vida. Se divide en dos categorías:

Documentación interna.

Documentación externa.

Documentación Interna:

Esta documentación cubre los aspectos del programa relativos a la sintaxis del lenguaje. Esta documentación está contenida en los comentarios, encerrados entre llaves o bien paréntesis / asteriscos. Algunos tópicos a considerar son:

Cabecera del programa (nombre del programador, fecha de la versión actual, breve descripción del programa).

Nombres significativos para describir identificadores.

Comentarios relativos a la función del programa como en todo, así como los módulos que comprenden el programa.

Claridad de estilo y formato [una sentencia por línea, identacion (sangrado)], líneas en blanco para separar módulos (procedimientos, funciones, unidades, etc.).

Comentarios significativos.

“La documentación interna consiste en un prólogo estándar para cada unidad de programa y unidad de compilación, los aspectos autodocumentados del código fuente y los comentarios internos intercalados en la porción ejecutable del código”.

Formato característico de los prólogos de subprogramas y unidades de compilación:

Nombre del autor:

Fecha de Compilación:

Función(es) realizada(s):

Algoritmos empleados;

Autor/fecha/propósito de la modificación:

Parámetros y modos:

aseveración de entrada:

aseveración de salida:

Variables globales:

Efectos colaterales:

Estructuras de datos principales:

Rutinas que invocan:

Rutinas invocadas:

Restricciones de tiempo:

Manejo de excepciones:

Suposiciones:

Convenciones sobre comentarios:

Minimícese la necesidad de comentarios inmensos en el código al usar:

Prologo estándar

Estructuras de programación estructurada

Buen estilo de programación

Nombres descriptivos del dominio del problema para tipos de datos definidos por el usuario, variables, parámetros formales literales de enumeración, subprogramas, archivos, etc.

Características de auto documentación del lenguaje de implementación, como excepciones definidas por el usuario, tipos de datos definidos por el usuario, encapsulado de datos, etc.

Adjúntese comentarios a los bloques de código que:

Realicen manipulaciones de datos importantes

Simulen construcciones de control estructuradas que manejen instrucciones goto

Realicen manejo de excepciones

Úsese la terminología del domino del problema en los comentarios

Empléese línea en blanco, delimitadora y sangrada para realizar los comentarios

Colóquense los comentarios a la extrema derecha para documentar cambios y revisiones

No se manejen comentarios largos y confusos para aclarar un código oscuro y complejo. Rescríbase el código.

Siempre es necesario asegurarse que el código y los comentarios correspondan con el otro, así como con los requisitos y especificaciones de diseño

Documentación externa:

Documentación ajena al programa fuente, que se suele incluir en un manual que acompaña al programa. La documentación externa debe incluir:

Listado actual del programa fuente, mapas de memoria, referencias cruzadas, etc.

Especificación del programa: documento que define el propósito y modo de funcionamiento del programa.

Diagrama de estructura que representa la organización jerárquica de los módulos que comprende el programa.

Explicaciones de fórmulas complejas.

Especificaciones de los datos a procesar: archivos externos incluyendo el formato de las estructuras de los registros, campos, etc.

Formatos de pantallas utilizados para interactuar con los usuarios.

Cualquier indicación especial que pueda servir a los programadores que deben mantener el programa.

**Manual del Usuario:**

El manual del usuario debe cubrir al menos los puntos:

Ordenes necesarias para cargar al programa en memoria desde el almacenamiento secundario (disco) y arrancar su funcionamiento.

Nombres de los archivos externos a lo que accede el programa

Formato de todos los mensajes de error o informes.

Opciones en el funcionamiento del programa.

Descripción detallada de la función realizada por el programa.

Descripción detallada, preferiblemente con ejemplos, de cualquier salida producida por el programa.

**Un buen estilo de Codificación:**

Estas normas son una guía de estilo para la codificación de programas. La experiencia ha demostrado que estas normas son beneficiosas para reducir errores semánticas y demandan cierta consistencia en la elección de nombres y la organización. Adicionalmente tienen por objeto asegurar la uniformidad de código entre los distintos programadores.

**Comentarios:**

La legibilidad de los programas aumenta considerablemente utilizando comentarios. Un comentario es un texto explicativo más o menos largo, situado en el programa e ignorado por el compilador. Los comentarios no se consideran (son invisibles) en la fase de compilación y ejecución, pero de importancia primordial en la fase de análisis, puesta a punto y mantenimiento.

Los comentarios son una parte importante de la documentación de un programa, ya que ayudan al programador y a otras personas a la comprensión del programa.

**Información sobre la historia de las versiones:**

Siempre es recomendable crear un encabezado al comienzo de cada archivo, en el cual se incluyan detalles tales como la fecha de creación, el autor, la versión del programa y una descripción general del contenido. Este bloque debe aparecer como un comentario

**Deshabilitar código con comentarios:**

Esta técnica se usa generalmente para diferenciar una sección del programa que no funciona correctamente. Una vez que hayamos solucionado el error simplemente borramos los marcadores de comentario y el código vuelve a estar activo nuevamente.

**Cadenas de Documentación:**

Todos los lenguajes permiten utilizar comentarios para documentar que realiza una función o un modulo, pero unos pocos van un paso más allá y permiten documentar las funciones de un modo tal que el mismo lenguaje puede proveer una ayuda interactiva en la fase de programación. Esto se realiza mediante una cadena de documentación.

**Sangrado:**

Esto es fuente de acalorados debates en los círculos de programación y parecería que cada programador tiene su propia idea acerca de cuál es el mejor manera de tabular el código. Se han realizado estudios que demuestran que al menos hay ciertos factores que realmente son importantes para una mayor legibilidad del programa y exceden un criterio simplemente estético.

**Los nombres de las variables:**

En general es recomendable que el nombre de la variable refleje claramente el contenido que almacena. Esto implica una lucha entre lo comprensible y lo sencillo. En general la mejor opción es utilizar nombre breve pero sensato. Un nombre muy largo se vuelve confuso y es difícil utilizarlo con consistencia a lo largo del programa. Cada variable local, con la excepción de las que tiene nombres realmente auto explicativos y los aburridos contadores de bucle, deben ser comentados cuando se declaran. Todas las variables globales deben ser comentadas sin excepción. Se debe realizar las especificaciones algebraicas de aquellos tipos que se definan por el programador.

**Funciones y Procedimientos:**

Comienzan con una descripción de su propósito obligatorio a no ser que sea obvio donde se especifican las variables de entrada el proceso y el resultado que devuelve. Los nombres de los parámetros deben ser explícitos. Para funciones muy genéricas nombres cortos pueden ser apropiados la función o el procedimiento realiza algún tipo de operación que puede dar un error el valor devuelto debe ser el código de error producido o 0 en caso satisfactorio. Los datos se devuelven a través de referencias.

**Convenciones de nombres:**

Las siguientes reglas especifican cuando utilizar letras mayúsculas o minúsculas en los identificadores:

Todos los nombres de variables y funciones y todos los miembros de estructuras van en minúsculas (pueden tener mayúsculas en el medio para identificar la separación de palabras dentro del identificador).

Todos los nombres de tipos, tipos abstractos de datos y clases deben empezar por mayúscula seguido de minúscula (pueden tener mayúsculas en el medio para identificar la separación de palabras dentro del identificador).

Para el nombre se deben utilizar únicamente las letras de la `a' a la `z' y este debe ser lo más clarificador posible.

Identacion y espacios en blanco:

Utilizar tabuladores fijados a tamaño constante (por ejemplo cada 3 columnas).

Utilizar líneas en blando libremente para separar partes lógicas distintas de una función.

Utilizar espacios en blando alrededor de cada operador binario.

Dejar espacios en blanco después (y no antes) de cada coma y punto y coma, pero no después de un nombre de función.

Todas las líneas deben encajar en 80 columnas. Si hay que romper una instrucción, añade un nivel de identacion para la continuación. Si es posible comenzar la línea identada con un operador.

Sangrar continuamente los diferentes niveles de código.

**Codificación es Pascal:**

En Pascal, los comentarios se encierran bien entre llaves o paréntesis acompañados por un asterisco. De hecho directivas de compilador también son comentarios.

**Uso de Mayúsculas:**

El compilador de Pascal no distingue entre mayúsculas y minúsculas. En general esto es bueno, ya que en otros lenguajes se producen muchos errores por grafía de mayúsculas incorrecta. Sin embargo esto conlleva algunas desventajas útiles. En primer lugar, usted debe ser consciente de que las dos formas de una letra son equivalentes y no utilizarlas para referirse a elementos distintos. En segundo lugar debería intentar ser coherente en el uso de mayúsculas y minúsculas, para hacer el código más legible.

Otro de los elementos que el compilador ignorar completamente son los espacios, las líneas nuevas y los espacios de tabulador. A estos elementos se les llama “espacios en blanco”. Un espacio en blanco solo sirve para hacer el código más legible. No afecta a la compilación.

Pascal le deja escribir una instrucción en varias líneas de código, extendiendo una instrucción larga en dos o más líneas. La desventaja de permitir esto es que debe acordarse de añadir un punto y como al final de cada instrucción o más exactamente acordarse de separar las instrucciones entre si.

**Impresión Bonita:**

La última sugerencia sobre el uso de espacio s en blanco se refiere al típico estilo de formato en Pascal llamado “Impresión Bonita” (pretty-printing). Esta regla es simple cada vez que necesite escribir una instrucción compuesta, haga una sangría de dos espacio a la derecha del resto de dicha instrucción. Una instrucción compuesta dentro de otra instrucción compuesta llevara una sangría de cuatro espacios y así sucesivamente.

**Resaltado Sintáctico:**

Para hacer más fácil el leer y escribir código pascal, el editor delphi tiene una cualidad llamada resaltado sintáctico. Dependiendo de la función en pascal de las palabras que usted introduce a través del editor, son representadas con distintos colores. Por defecto las palabras clave están en legra negrita, cadena de texto y comentarios están en color.

**DIRECTRICES DE DOCUMENTACION:**

La programación por computadora incluye el código fuente de un sistema y todos los documentos de apoyo generados durante el análisis, diseño, instrumentación, pruebas y mantenimiento del sistema. La documentación interna incluye prólogos estándar para unidades de compilación y subprogramas, los aspectos autodocumentados del código fuente, y los comentarios internos, incrustados en el código fuente. Las notas de cada unidad proporcionan mecanismos para organizar las actividades del trabajo y esfuerzos de documentación de cada programador. Esta sección describe algunos aspectos de los documentos de apoyo, el uso de notas de cada unidad del programa y algunos principios generales para la documentación interna del código fuente.

**Documentos de apoyo:**

Las especificaciones de requisitos, documentos de diseño, planes de prueba, manuales de usuario, instrucciones de instalación y los reportes de mantenimiento son ejemplos de documentos de apoyo. Estos documentos son los productos que resultan del desarrollo y mantenimiento sistemático de la programación. Las herramientas, técnicas y notaciones para generar y dar mantenimiento a estos documentos se analizan en todo este texto.

Un enfoque sistemático al desarrollo de la programación garantiza que los documentos de apoyo se desarrollen de una manera ordenada y que esos documentos se encuentren disponibles cuando se necesiten. En el enfoque adecuado para el desarrollo de la programación, la preparación de documentos de apoyo normalmente se difiere hasta que se termine la instrumentación del sistema. Debido a restricciones de tiempo y falta de motivación, los documentos que se generan de esta manera normalmente son inadecuados para soportar actividades de prueba, entrenamiento, modificaciones y mantenimiento.

Los documentos de apoyo de calidad inferior a la estándar que no están disponibles cuando se necesitan, son una fuerte indicación de problemas con el proceso empleado en el desarrollo y mantenimiento de la programación. Estos documentos deben desarrollarse como un producto natural paralelo al proceso de desarrollo. Las necesidades y restricciones del cliente se registran en la especificación de requisitos, los requisitos proporcionan el marco de trabajo para el diseño estructural, el diseño detallado se desarrolla a partir del diseño estructural, el código fuente se desarrolla a partir de los diseños estructurales y detallados. Los planes de prueba, manuales de usuario, programas de entrenamiento, instrucciones de instalación y procedimientos de mantenimiento evolucionan a través de todo el ciclo de desarrollo. La calidad, duración y utilidad de los documentos de apoyo son las principales medidas de la salud y bondad de un proyecto de programación.

**Notas de unidad de programa:**

Una unidad de programa es una unidad de código fuente que es desarrollada y/o mantenida por una persona; esa persona es la responsable de la unidad. En un sistema bien diseñado, una unidad de programa es un subprograma o grupo de subprogramas que cumplen una función bien definida o forman un subsistema bien definido. Una unidad de programa también es lo suficientemente pequeño y modular que puede se probada totalmente en forma aislada por el programador que la desarrolla o modifica. Las notas de cada unidad de programa son utilizadas por cada programador para organizar sus actividades de trabajo y para conservar la documentación de sus unidades de programa.

Un cuaderno de notas de cada unidad de programa (también conocido como carpeta de desarrollo de unidad) constan de una portada y varias secciones. La portada es la tabla de contenidos y la hoja de avisos de terminación de los diversos logros asociados con la unidad del programa. La figura ilustra la portada de un cuaderno de notas de cada unidad de programa. El mantenimiento del cuaderno de notas es responsabilidad del programador asignado a la unidad de programa. El cuaderno de notas permanece con la unidad de programa durante todo su tiempo de vida y pasa de programador a programador a medida que se van transfiriendo las responsabilidades.

Las secciones dentro de un cuaderno de notas de la unidad corresponden a las diversas fases del ciclo de vida de la unidad. La portada se usa para registrar las fechas proyectada y real de los logros y el aviso de terminación hecho por un revisor indica la culminación satisfactoria de una fase de ciclo de vida. Los revisores normalmente son líderes de grupo o administradores de proyecto que tienen la responsabilidad del sistema total o de un subsistema significativo.

La sección de requisitos de un cuaderno de notas de unidad contiene solo las especificaciones concernientes de esa unidad de programa particular. Tanto la versión inicial como se convino con el cliente, como las copias de las modificaciones subsecuentes a los requisitos se conservan en el cuaderno de notas

Las secciones de diseño estructural y detallado del cuaderno de notas contienen los papeles de trabajo y las especificaciones de diseño finales. Los papeles de trabajo deben organizarse para indicar porque se eligieron diversas alternativas y otras se rechazaron. Esta información puede ser valiosa para guiar las modificaciones subsecuentes.

El plan de pruebas de la unidad contiene una descripción del enfoque que será utilizado en la prueba de la unidad, la configuración de prueba, los casos de prueba actuales, los resultados esperados para cada caso de prueba y los criterios de terminación de las pruebas de la unidad. El plan de pruebas de la unidad se desarrolla junto con el diseño e instrumentación de la unidad.

La sección que contiene el código fuente de la unidad tiene un listado de la versión actual de la unidad y listados de las versiones previas significativas. No se espera que se conserven todas las corridas sintácticas y de depuración, sino los listados de las primeras versiones que existieron previas a las modificaciones significativas, particularmente en la fase de mantenimiento del ciclo de vida.

La sección de resultados de las pruebas contiene los resultados de las corridas de prueba y un análisis de ellos en términos de los resultados esperados. De nuevo no deben conservarse todas las corridas de depuración, sino solo las corridas de prueba significativas (tanto las exitosas como las que no lo son).

Los reportes de problemas se llenan después de que el sistema cae bajo control de la configuración. En algún punto del ciclo de vida (normalmente durante las pruebas de aceptación, o en la liberación del producto) un sistema cae bajo control de la configuración y cualesquiera modificaciones subsecuentes deben ser revisadas y aprobadas por una junta de control de cambios. Los reportes de problemas son el mecanismo que se usa ordinariamente para iniciar un cambio. El reporte de problemas puede describir una situación de error o una modificación deseada. Una copia de cada reporte de problemas y su disposición deben mantenerse en el cuaderno de notas de unidad de las unidades afectadas. Pasado el tiempo los reportes de problemas mostraran la historia de las unidades. En algunos casos la revisión de los reportes de problemas revelaran los errores recurrentes en ciertas unidades o colecciones de unidades. A veces es recomendable rescribir las unidades problemáticas en vez de continuar reparándolas.

**Programación Orientada a Objetos:**

Un programa escrito con un lenguaje orientado a objetos es un conjunto de objetos que interactúan entre si mandándose mensajes.

Los elementos que interactúan en la POO son:

**Objetos:**Son el elementos fundamental de la POO, son entidades que poseen un estado interno y un comportamiento, es lo equivalente a un dato abstracto. Un objeto puede ser cualquier entidad del mundo real por Ej.: Automóviles en una simulación de tráfico, Componentes electrónicos en un programa de diseño de circuitos, etc.

**Mensajes:** Es una petición de un objeto a otro para que este se comporte de una determinada manera, ejecutando uno de sus métodos.

**Métodos:** es un programa que está asociado un objeto determinado cuya ejecución solo puede desencadenarse a través de un mensaje recibido por este o por sus descendientes

**Clases:** Es un tipo definido por el usuario que determina las estructuras de datos y las operaciones asociadas con ese tipo. Cada objeto pertenece a una clase y recibe de ella su funcionalidad. Es el primer nivel de abstracción de datos: definimos estructuras, comportamientos y tenemos ocultamiento. Un objeto es una caja negra cuya parte interna permanece oculta. La información contenida en el objeto solo puede ser accedida por la ejecución de los métodos correspondientes.

**Instancia de Clases:**

Cada vez que se construye un objeto se está creando una instancia de esa clase. Una instancia es un objeto individualizado por los valores que tomen sus atributos. Es otro aspecto de la abstracción de datos.

**Herencia:**

El segundo nivel de abstracción consiste en agrupar las clases en jerarquías de clases

(Definiendo Sub y Súper clases), de forma tal que una clase A herede todas las propiedades de su superclase B (suponiendo que tiene una). Solo se necesita definir los atributos y comportamiento que especializan la definición, el resto se Hereda.

Tipos de Herencia:

**Simple:** Cada clase puede tener una sola Súper-Clase (Esquema de Árbol)

**Múltiple:** Una clase puede heredar de varias Súper-Clases.

**Estricta:** Una Sub-Clase no puede redefinir en forma contradictoria atributos definidos en la Súper-Clase.

**No Estricta:** Alguna Sub-Clase no quiere heredar algún atributo de la Súper-Clase.

**Polimorfismo:**

Es la capacidad que tienen los objetos de distintas clases de responder a mensajes con el mismo nombre.

Ventajas de la POO:

* Más natural
* Modularidad
* Extensibilidad
* Reusabilidad

**CARACTERISTICAS DEL LENGUAJE C**  
  
     El lenguaje C se conoce como un lenguaje compilado. Existen dos tipos de lenguaje: interpretados y compilados. Los interpretados son aquellos que necesitan del código fuente para funcionar (P.ej: Basic). Los compilados convierten el código fuente en un fichero objeto y éste en un fichero ejecutable. Este es el caso del lenguaje C.  
     Podemos decir que el lenguaje C es un lenguaje de nivel medio, ya que combina elementos de lenguaje de alto nivel con la funcionalidad del lenguaje ensamblador. Es un lenguaje estructurado, ya que permite crear procedimientos en bloques dentro de otros procedimientos. Hay que destacar que el C es un lenguaje portable, ya que permite utilizar el mismo código en diferentes equipos y sistemas informáticos: el lenguaje es independiente de la arquitectura de cualquier máquina en particular.  
     Por último solo queda decir que el C es un lenguaje relativamente pequeño; se puede describir en poco espacio y aprender rápidamente. Este es sin duda el objetivo de éste curso. No pretende ser un completo manual de la programación, sino una base útil para que cualquiera pueda introducirse en este apasionante mundo.  
     Aunque en principio cualquier compilador de C es válido, para seguir este curso se recomienda utilizar el compilador Turbo C/C++ o bien el [Borland C++ 5.0](http://www.borland.com/bcppbuilder/freecompiler).

**ESTRUCTURA DE UN PROGRAMA EN C**

 Todo programa en C consta de una o más funciones, una de las cuales se llama main. El programa comienza en la función main, desde la cual es posible llamar a otras funciones.  
 Cada función estará formada por la cabecera de la función, compuesta por el nombre de la misma y la lista de argumentos (si los hubiese), la declaración de las variables a utilizar y la secuencia de sentencias a ejecutar.  
  
     Ejemplo:

Declaraciones globales

Main( ) {

Variables locales

Bloque

}

funcion1 ( ) {

Variables locales

Bloque

}  
  
     A la hora de programar es conveniente añadir comentarios (cuantos más mejor) para poder saber que función tiene cada parte del código, en caso de que no lo utilicemos durante algún tiempo. Además facilitaremos el trabajo a otros programadores que puedan utilizar nuestro archivo fuente.  
     Para poner comentarios en un programa escrito en C usamos los símbolos /\* y \*/:

/\* Este es un ejemplo de comentario \*/

/\* Un comentario también puede

Estar escrito en varias líneas \*/

     El símbolo /\* se coloca al principio del comentario y el símbolo \*/ al final.  
     El comentario, contenido entre estos dos símbolos, no será tenido en cuenta por el compilador.

**Palabras clave**  
 Existen una serie de indicadores reservados, con una finalidad determinada, que no podemos utilizar como identificadores.  
     A continuación vemos algunas de estas palabras clave:

Char int float double if

Else do while for switch

short long extern static default

Continue break register sizeof typedef

**Identificadores**  
     Un identificador es el nombre que damos a las variables y funciones. Está formado por una secuencia de letras y dígitos, aunque también acepta el carácter de subrayado \_. Por contra no acepta los acentos ni la ñ/Ñ.  
     El primer caracter de un identificador no puede ser un número, es decir que debe ser una letra o el símbolo \_.  
     Se diferencian las mayúsculas de las minúsculas, así num, Num y nuM son distintos identificadores.  
     A continuación vemos algunos ejemplos de identificadores válidos y no válidos:

Válidos No válidos

\_num 1num

var1 número2

fecha\_nac año\_nac

**TIPOS DE DATOS**

**Tipos**  
 En 'C' existen básicamente cuatro tipos de datos, aunque como se verá después, podremos definir nuestros propios tipos de datos a partir de estos cuatro. A continuación se detalla su nombre, el tamaño que ocupa en memoria y el rango de sus posibles valores.

TIPO Tamaño Rango de valores

Char 1 byte -128 a 127

int 2 bytes -32768 a 32767

Float 4 bytes 3'4 E-38 a 3'4 E+38

Double 8 bytes 1'7 E-308 a 1'7 E+308

**Calificadores de tipo**  
 Los calificadores de tipo tienen la misión de modificar el rango de valores de un determinado tipo de variable. Estos calificadores son cuatro:

**SIGNED**  
  
Le indica a la variable que va a llevar signo. Es el utilizado por defecto.

Tamaño rango de valores

Signed char 1 byte -128 a 127

Signed int 2 bytes -32768 a 32767

Unsigned  
  
Le indica a la variable que no va a llevar signo (valor absoluto).

Tamaño rango de valores

Unsigned char 1 byte 0 a 255

Unsigned int 2 bytes 0 a 65535

Short  
  
Rango de valores en formato corto (limitado). Es el utilizado por defecto.

tamaño rango de valores

Short char 1 byte -128 a 127

Short int 2 bytes -32768 a 32767

Long  
  
Rango de valores en formato largo (ampliado).

Tamaño rango de valores

Long int 4 bytes -2.147.483.648 a 2.147.483.647

Long double 10 bytes -3'36 E-4932 a 1'18 E+4932

También es posible combinar calificadores entre sí:

Signed long int = long int = long

Unsigned long int = unsigned long 4 bytes 0 a 4.294.967.295 (El mayor entero permitido en 'C')

**Las variables**  
     Una variable es un tipo de dato, referenciado mediante un identificador (que es el nombre de la variable). Su contenido podrá ser modificado a lo largo del programa.  
     Una variable sólo puede pertenecer a un tipo de dato. Para poder utilizar una variable, primero tiene que ser declarada:

[Calificador] <Tipo> <nombre>

     Es posible inicializar y declarar más de una variable del mismo tipo en la misma sentencia:

[Calificador] <Tipo> <nombre1>,<nombre2>=<valor>,<nombre3>=<valor>,<nombre4>

Ejemplo:

/\* Uso de las variables \*/

#include <stdio.h>

main() /\* Suma dos valores \*/

{

int num1=4, num2,num3=6;

Printf ("El valor de num1 es %d", num1);

Printf ("\El valor de num3 es %d", num3);

num2=num1+num3;

Printf ("\nnum1 + num3 = %d", num2);

}

**¿Dónde se declaran?**  
  
     Las variables pueden ser de dos tipos según el lugar en que las declaremos: globales o locales.  
     La variable global se declara antes de la main( ). Puede ser utilizada en cualquier parte del programa y se destruye al finalizar éste.  
     La variable local se declara después de la main ( ), en la función en que vaya a ser utilizada. Sólo existe dentro de la función en que se declara y se destruye al finalizar dicha función.  
     El identificador (nombre de la variable) no puede ser una palabra clave y los caracteres que podemos utilizar son las letras: a-z y A-Z (ojo! la ñ o Ñ no está permitida), los números: 0-9 y el símbolo de subrayado \_. Además hay que tener en cuenta que el primer carácter no puede ser un número.

Ejemplo:

/\* Declaración de variables \*/

#include <stdio.h>

Int a;

Main () /\* Muestra dos valores \*/

{

Int b=4;

Printf ("b es local y vale %d", b);

a=5;

Printf ("\na es global y vale %d",a);

}

**Constantes**  
     Al contrario que las variables, las constantes mantienen su valor a lo largo de todo el programa.  
     Para indicar al compilador que se trata de una constante, usaremos la directiva #define:

#define <identificador> <valor>

     Observa que no se indica el punto y coma de final de sentencia ni tampoco el tipo de dato.  
     La directiva #define no sólo nos permite sustituir un nombre por un valor numérico, sino también por una cadena de caracteres.  
     El valor de una constante no puede ser modificado de ninguna manera.

Ejemplo:

/\* Uso de las constantes \*/

#include <stdio.h>

#define pi 3.1416

#define escribe printf

Main() /\* Calcula el perímetro \*/

{

Int r;

Escribe ("Introduce el radio: ");

Scanf ("%d",&r);

Escribe ("El perímetro es: %f",2\*pi\*r);

}

**Secuencias de escape**  
Ciertos caracteres no representados gráficamente se pueden representar mediante lo que se conoce como secuencia de escape.  
     A continuación vemos una tabla de las más significativas:

\n salto de línea

\b retroceso

\t tabulación horizontal

\v tabulación vertical

\\ contra barra

\f salto de página

\' apóstrofe

\" comillas dobles

\0 fin de una cadena de caracteres

Ejemplo:

/\* Uso de las secuencias de escape \*/

#include <stdio.h>

Main() /\* Escribe diversas sec. De escape \*/

{

Printf ("Me llamo \"Nemo\" el grande");

Printf ("\nDirección: C\\ Mayor 25");

Printf ("\nHa salido la letra \'L\'");

Printf ("\nRetroceso\b");

Printf ("\n\tEsto ha sido todo");

}

**Inclusión de ficheros**  
  
     En la programación en C es posible utilizar funciones que no estén incluidas en el propio programa. Para ello utilizamos la directiva#include, que nos permite añadir librerías o funciones que se encuentran en otros ficheros a nuestro programa.  
     Para indicar al compilador que vamos a incluir ficheros externos podemos hacerlo de dos maneras (siempre antes de las declaraciones).  
     1. Indicándole al compilador la ruta donde se encuentra el fichero.

#include "misfunc.h"

#include "c:\includes\misfunc.h"

     2. Indicando que se encuentran en el directorio por defecto del compilador.

#include <misfunc.h>

**OPERADORES ARITMETICOS Y DE ASIGNACION**

A continuación se explican los tipos de operadores (aritméticos y de asignación) que permiten realizar operaciones matemáticas en lenguaje C.

**Operadores aritméticos**  
  
     Existen dos tipos de operadores aritméticos:  
     Los binarios:

+ Suma

- Resta

\* Multiplicación

/ División

% Módulo (resto)

     y los unarios:

++ Incremento (suma 1)

- - Decremento (resta 1)

- Cambio de signo

     Su sintaxis es:

Binarios:

<variable1><operador><variable2>

Unarios:

<Variable><operador> y al revés, <operador><variable>.

Ejemplo:

/\* Uso de los operadores aritméticos \*/

#include <stdio.h>

Main () /\* Realiza varias operaciones \*/

{

Int a=1,b=2,c=3,r;

r=a+b;

Printf ("%d + %d = %d\n",a,b,r);

r=c-a;

Printf ("%d - %d = %d\n", c, a, r);

b++;

Printf ("b + 1 = %d",b);

}

**Operadores de asignación**

 La mayoría de los operadores aritméticos binarios explicados en el capítulo anterior tienen su correspondiente operador de asignación:

= Asignación simple

+= Suma

-= Resta

\*= Multiplicación

/= División

%= Módulo (resto)

     Con estos operadores se pueden escribir, de forma más breve, expresiones del tipo:

n=n+3 se puede escribir n+=3  
k=k\*(x-2) lo podemos sustituir por k\*=x-2

Ejemplo:

/\* Uso de los operadores de asignación \*/

#include <stdio.h>

Main() /\* Realiza varias operaciones \*/

{

int a=1,b=2,c=3,r;

a+=5;

printf("a + 5 = %d\n",a);

c-=1;

printf("c - 1 = %d\n",c);

b\*=3;

printf("b \* 3 = %d",b);

}

**Jerarquía de los operadores**  
Será importante tener en cuenta la precedencia de los operadores a la hora de trabajar con ellos:

( ) Mayor precedencia

++, - -

\*, /, %

+, - Menor precedencia

     Las operaciones con mayor precedencia se realizan antes que las de menor precedencia.  
     Si en una operación encontramos signos del mismo nivel de precedencia, dicha operación se realiza de izquierda a derecha. A continuación se muestra un ejemplo sobre ello:

a\*b+c/d-e

1. a\*b resultado = x

2. c/d resultado = y

3. x+y resultado = z

Fijarse que la multiplicación se resuelve antes que la división ya que está situada más a la izquierda en la operación. Lo mismo ocurre con la suma y la resta.

Ejemplo:

/\* Jerarquía de los operadores \*/

#include <stdio.h>

main() /\* Realiza una operación \*/

{

int a=6,b=5,c=4,d=2,e=1,x,y,z,r;

x=a\*b;

printf("%d \* %d = %d\n",a,b,x);

y=c/d;

printf("%d / %d = %d\n",c,d,y);

z=x+y;

printf("%d + %d = %d\n",x,y,z);

r=z-e;

printf("%d = %d",r,a\*b+c/d-e);

}

**SALIDA / ENTRADA**

**Sentencia printf ( )**  
     La rutina printf permite la aparición de valores numéricos, caracteres y cadenas de texto por pantalla.  
     El prototipo de la sentencia printf es el siguiente:

Printf (control, arg1, arg2...);

     En la cadena de control indicamos la forma en que se mostrarán los argumentos posteriores. También podemos introducir una cadena de texto (sin necesidad de argumentos), o combinar ambas posibilidades, así como secuencias de escape.  
     En el caso de que utilicemos argumentos deberemos indicar en la cadena de control tantos modificadores como argumentos vayamos a presentar.  
     El modificador está compuesto por el carácter % seguido por un carácter de conversión, que indica de qué tipo de dato se trata.

Ejemplo:

/\* Uso de la sentencia printf() 1. \*/

#include <stdio.h>

main() /\* Saca por pantalla una suma \*/

{

int a=20,b=10;

printf("El valor de a es %d\n",a);

printf("El valor de b es %d\n",b);

printf("Por tanto %d+%d=%d",a,b,a+b);

}

Los modificadores más utilizados son:

%c Un único caracter

%d Un entero con signo, en base decimal

%u Un entero sin signo, en base decimal

%o Un entero en base octal

%x Un entero en base hexadecimal

%e Un número real en coma flotante, con exponente

%f Un número real en coma flotante, sin exponente

%s Una cadena de caracteres

%p Un puntero o dirección de memoria

Ejemplo:

/\* Uso de la sentencia printf() 2. \*/

#include <stdio.h>

main() /\* Modificadores 1 \*/

{

char cad[]="El valor de";

int a=-15;

unsigned int b=3;

float c=932.5;

printf("%s a es %d\n",cad,a);

printf("%s b es %u\n",cad,b);

printf("%s c es %e o %f",cad,c,c);

}

El formato completo de los modificadores es el siguiente:

% [signo] [longitud] [.precisión] [l/L] conversión

     Signo: indicamos si el valor se ajustará a la izquierda, en cuyo caso utilizaremos el signo menos, o a la derecha ( por defecto ).  
     Longitud: especifica la longitud máxima del valor que aparece por pantalla. Si la longitud es menor que el número de dígitos del valor, éste aparecerá ajustado a la izquierda.  
  
     Precisión: indicamos el número máximo de decimales que tendrá el valor.  
     l/L: utilizamos l cuando se trata de una variable de tipo long y L cuando es de tipo double.

Ejemplo:

/\* Uso de la sentencia printf() 3. \*/

#include <stdio.h>

main() /\* Modificadores 2 \*/

{

char cad[ ]="El valor de";

int a=25986;

long int b=1976524;

float c=9.57645;

printf("%s a es %9d\n",cad,a);

printf("%s b es %ld\n",cad,b);

printf("%s c es %.3f",cad,c);

}

**Sentencia scanf ( )**  
  
     La rutina scanf permite entrar datos en la memoria del ordenador a través del teclado.  
     El prototipo de la sentencia scanf es el siguiente:

Scanf (control, arg1, arg2...);

     En la cadena de control indicaremos, por regla general, los modificadores que harán referencia al tipo de dato de los argumentos. Al igual que en la sentencia printf los modificadores estarán formados por el carácter % seguido de un carácter de conversión. Los argumentos indicados serán, nuevamente, las variables.  
     La principal característica de la sentencia scanf es que necesita saber la posición de la memoria del ordenador en que se encuentra la variable para poder almacenar la información obtenida. Para indicarle esta posición utilizaremos el símbolo ampersand ( & ), que colocaremos delante del nombre de cada variable. ( Esto no será necesario en los arrays ).

Ejemplo:

/\* Uso de la sentencia scanf(). \*/

#include <stdio.h>

main() /\* Solicita dos datos \*/

{

char nombre[10];

int edad;

printf("Introduce tu nombre: ");

scanf("%s",nombre);

printf("Introduce tu edad: ");

scanf("%d",&edad);

}

**OPERADORES RELACIONALES**

Los operadores relacionales se utilizan para comparar el contenido de dos variables.  
     En C existen seis operadores relacionales básicos:

> Mayor que

< Menor que

>= Mayor o igual que

<= Menor o igual que

== Igual que

!= Distinto que

     El resultado que devuelven estos operadores es 1 para Verdadero y 0 para Falso.  
     Si hay más de un operador se evalúan de izquierda a derecha. Además los operadores == y != están por debajo del resto en cuanto al orden de precedencia.

Ejemplo:

/\* Uso de los operadores relacionales. \*/

#include <stdio.h>

main() /\* Compara dos números entre ellos \*/

{

int a,b;

printf("Introduce el valor de A: ");

scanf("%d",&a);

printf("Introduce el valor de B: ");

scanf("%d",&b);

if(a>b)

printf("A es mayor que B");

else if(a<b)

printf("B es mayor que A");

else

printf("A y B son iguales");

}

 7.- SENTENCIAS CONDICIONALES

Este tipo de sentencias permiten variar el flujo del programa en base a unas determinadas condiciones.  
     Existen varias estructuras diferentes:  
  
 **Estructura IF...ELSE**  
  
  **Sintaxis:**

if (condición) sentencia;

     La sentencia solo se ejecuta si se cumple la condición. En caso contrario el programa sigue su curso sin ejecutar la sentencia.  
     Otro formato:

if (condición) sentencia1;

else sentencia2;

     Si se cumple la condición ejecutará la sentencia1, sinó ejecutará la sentencia2. En cualquier caso, el programa continuará a partir de lasentencia2.

Ejemplo:

/\* Uso de la sentencia condicional IF. \*/

#include <stdio.h>

main() /\* Simula una clave de acceso \*/

{

int usuario,clave=18276;

printf("Introduce tu clave: ");

scanf("%d",&usuario);

if(usuario==clave)

printf("Acceso permitido");

else

printf("Acceso denegado");

}

Otro formato:

if (condición) sentencia1;

else if (condición) sentencia2;

else if (condición) sentencia3;

else sentencia4;

     Con este formato el flujo del programa únicamente entra en una de las condiciones. Si una de ellas se cumple, se ejecuta la sentencia correspondiente y salta hasta el final de la estructura para continuar con el programa.  
     Existe la posibilidad de utilizar llaves para ejecutar más de una sentencia dentro de la misma condición.

Ejemplo:

/\* Uso de la sentencia condicional ELSE...IF. \*/

#include <stdio.h>

main() /\* Escribe bebé, niño o adulto \*/

{

int edad;

printf("Introduce tu edad: ");

scanf("%d",&edad);

if (edad<1)

printf("Lo siento, te has equivocado.");

else if (edad<3) printf("Eres un bebé");

else if (edad<13) printf("Eres un niño");

else printf("Eres adulto");

}

**ESTRUCTURA SWITCH**  
  
     Esta estructura se suele utilizar en los menús, de manera que según la opción seleccionada se ejecuten una serie de sentencias.  
     Su sintaxis es:

switch (variable){

case contenido\_variable1:

sentencias;

break;

case contenido\_variable2:

sentencias;

break;

default:

sentencias;

}

     Cada case puede incluir una o más sentencias sin necesidad de ir entre llaves, ya que se ejecutan todas hasta que se encuentra la sentencia BREAK. La variable evaluada sólo puede ser de tipo entero o caracter. default ejecutará las sentencias que incluya, en caso de que la opción escogida no exista.

Ejemplo:

/\* Uso de la sentencia condicional SWITCH. \*/

#include <stdio.h>

main() /\* Escribe el día de la semana \*/

{

int dia;

printf("Introduce el día: ");

scanf("%d",&dia);

switch(dia){

case 1: printf("Lunes"); break;

case 2: printf("Martes"); break;

case 3: printf("Miércoles"); break;

case 4: printf("Jueves"); break;

case 5: printf("Viernes"); break;

case 6: printf("Sábado"); break;

case 7: printf("Domingo"); break;

}

}

**OPERADORES LOGICOS**

Los operadores lógicos básicos son tres:

&& AND

|| OR

! NOT (El valor contrario)

     Estos operadores actúan sobre expresiones lógicas. Permiten unir expresiones lógicas simples formando otras más complejas.

ejemplo:

/\* Uso de los op. lógicos AND,OR,NOT. \*/

#include <stdio.h>

main() /\* Compara un número introducido \*/

{

int numero;

printf("Introduce un número: ");

scanf("%d",&numero);

if(!(numero>=0))

printf("El número es negativo");

else if((numero<=100)&&(numero>=25))

printf("El número está entre 25 y 100");

else if((numero<25)||(numero>100))

printf("El número no está entre 25 y 100");

}

**BUCLES**

Los bucles son estructuras que permiten ejecutar partes del código de forma repetida mientras se cumpla una condición.  
     Esta condición puede ser simple o compuesta de otras condiciones unidas por operadores lógicos.  
  
**Sentencia WHILE**  
  
     Su sintaxis es:

while (condición) sentencia;

     Con esta sentencia se controla la condición antes de entrar en el bucle. Si ésta no se cumple, el programa no entrará en el bucle.  
     Naturalmente, si en el interior del bucle hay más de una sentencia, éstas deberán ir entre llaves para que se ejecuten como un bloque.

Ejemplo:

/\* Uso de la sentencia WHILE. \*/

#include <stdio.h>

main() /\* Escribe los números del 1 al 10 \*/

{

int numero=1;

while(numero<=10)

{

printf("%d\n",numero);

numero++;

}

}

**Sentencia DO...WHILE**  
  
     Su sintaxis es:

do{

sentencia1;

sentencia2;

}while (condición);

     Con esta sentencia se controla la condición al final del bucle. Si ésta se cumple, el programa vuelve a ejecutar las sentencias del bucle.  
     La única diferencia entre las sentencias while y do...while es que con la segunda el cuerpo del bucle se ejecutará por lo menos una vez.

Ejemplo:

/\* Uso de la sentencia DO...WHILE. \*/

#include <stdio.h>

main() /\* Muestra un menú si no se pulsa 4 \*/

{

char seleccion;

do{

printf("1.- Comenzar\n");

printf("2.- Abrir\n");

printf("3.- Grabar\n");

printf("4.- Salir\n");

printf("Escoge una opción: ");

seleccion=getchar();

switch(seleccion){

case '1':printf("Opción 1");

break;

case '2':printf("Opción 2");

break;

case '3':printf("Opción 3");

}

}while(seleccion!='4');

}

**Sentencia FOR**  
  
     Su sintaxis es:

for (inicialización;condición;incremento){

sentencia1;

sentencia2;

}

     La inicialización indica una variable (variable de control) que condiciona la repetición del bucle. Si hay más, van separadas por comas:

for (a=1,b=100;a!=b;a++,b- -){

  El flujo del bucle FOR transcurre de la siguiente forma:

Ejemplo:

/\* Uso de la sentencia FOR. \*/

#include <stdio.h>

main() /\* Escribe la tabla de multiplicar \*/

{

int num,x,result;

printf("Introduce un número: ");

scanf("%d",&num);

for (x=0;x<=10;x++){

result=num\*x;

printf("\n%d por %d = %d\n",num,x,result);

}

}

**Sentencia BREAK**  
  
     Esta sentencia se utiliza para terminar la ejecución de un bucle o salir de una sentencia SWITCH.

**Sentencia CONTINUE**  
  
     Se utiliza dentro de un bucle. Cuando el programa llega a una sentencia CONTINUE no ejecuta las líneas de código que hay a continuación y salta a la siguiente iteración del bucle.  
     Y aquí termina el capítulo dedicado a los bucles. Existe otra sentencia, GOTO, que permite al programa saltar hacia un punto identificado con una etiqueta, pero el buen programador debe prescindir de su utilización. Es una sentencia muy mal vista en la programación en 'C'.

Ejemplo:

/\* Uso de la sentencia CONTINUE. \*/

#include <stdio.h>

main() /\* Escribe del 1 al 100 menos el 25 \*/

{

int numero=1;

while(numero<=100)

{

if (numero==25)

{

numero++;

continue;

}

printf("%d\n",numero);

numero++;

}

}

**FUNCIONES**

**Tiempo de vida de los datos**  
     Según el lugar donde son declaradas puede haber dos tipos de variables.  
     Globales: las variables permanecen activas durante todo el programa. Se crean al iniciarse éste y se destruyen de la memoria al finalizar. Pueden ser utilizadas en cualquier función.  
     Locales: las variables son creadas cuando el programa llega a la función en la que están definidas. Al finalizar la función desaparecen de la memoria.  
     Si dos variables, una global y una local, tienen el mismo nombre, la local prevalecerá sobre la global dentro de la función en que ha sido declarada.  
     Dos variables locales pueden tener el mismo nombre siempre que estén declaradas en funciones diferentes.

Ejemplo:

/\* Variables globales y locales. \*/

#include <stdio.h>

int num1=1;

main() /\* Escribe dos cifras \*/

{

int num2=10;

printf("%d\n",num1);

printf("%d\n",num2);

}

**Funciones**  
     Las funciones son bloques de código utilizados para dividir un programa en partes más pequeñas, cada una de las cuáles tendrá una tarea determinada.  
     Su sintaxis es:

tipo\_función nombre\_función (tipo y nombre de argumentos)

{

bloque de sentencias

}

     tipo\_función: puede ser de cualquier tipo de los que conocemos. El valor devuelto por la función será de este tipo. Por defecto, es decir, si no indicamos el tipo, la función devolverá un valor de tipo entero ( int ). Si no queremos que retorne ningún valor deberemos indicar el tipo vacío ( void ).  
     nombre\_función: es el nombre que le daremos a la función.  
     tipo y nombre de argumentos: son los parámetros que recibe la función. Los argumentos de una función no son más que variables locales que reciben un valor. Este valor se lo enviamos al hacer la llamada a la función. Pueden existir funciones que no reciban argumentos.  
     bloque de sentencias: es el conjunto de sentencias que serán ejecutadas cuando se realice la llamada a la función.  
     Las funciones pueden ser llamadas desde la función main o desde otras funciones. Nunca se debe llamar a la función main desde otro lugar del programa. Por último recalcar que los argumentos de la función y sus variables locales se destruirán al finalizar la ejecución de la misma.

**Declaración de las funciones**  
  
     Al igual que las variables, las funciones también han de ser declaradas. Esto es lo que se conoce como prototipo de una función. Para que un programa en C sea compatible entre distintos compiladores es imprescindible escribir los prototipos de las funciones.  
     Los prototipos de las funciones pueden escribirse antes de la función main o bién en otro fichero. En este último caso se lo indicaremos al compilador mediante la directiva #include.  
     En el ejemplo adjunto podremos ver la declaración de una función ( prototipo ). Al no recibir ni retornar ningún valor, está declarada comovoid en ambos lados. También vemos que existe una variable global llamada num. Esta variable es reconocible en todas las funciones del programa. Ya en la función main encontramos una variable local llamada num. Al ser una variable local, ésta tendrá preferencia sobre la global. Por tanto la función escribirá los números 10 y 5.

Ejemplo:

/\* Declaración de funciones. \*/

#include <stdio.h>

void funcion(void); /\* prototipo \*/

int num=5; /\* variable global \*/

main() /\* Escribe dos números \*/

{

int num=10; /\* variable local \*/

printf("%d\n",num);

funcion(); /\* llamada \*/

}

void funcion(void)

{

printf("%d\n",num);

}

**Paso de parámetros a una función**  
  
     Como ya hemos visto, las funciones pueden retornar un valor. Esto se hace mediante la instrucción return, que finaliza la ejecución de la función, devolviendo o no un valor.  
     En una misma función podemos tener más de una instrucción return. La forma de retornar un valor es la siguiente:

return ( valor o expresión );

     El valor devuelto por la función debe asignarse a una variable. De lo contrario, el valor se perderá.  
     En el ejemplo puedes ver lo que ocurre si no guardamos el valor en una variable. Fíjate que a la hora de mostrar el resultado de la suma, en el printf, también podemos llamar a la función.

Ejemplo:

/\* Paso de parámetros. \*/

#include <stdio.h>

int suma(int,int); /\* prototipo \*/

main() /\* Realiza una suma \*/

{

int a=10,b=25,t;

t=suma(a,b); /\* guardamos el valor \*/

printf("%d=%d",suma(a,b),t);

suma(a,b); /\* el valor se pierde \*/

}

int suma(int a,int b)

{

return (a+b);

}

Ahora veremos lo que se conoce como paso de parámetros.  
     Existen dos formas de enviar parámetros a una función:  
     Por valor: cualquier cambio que se realice dentro de la función en el argumento enviado, NO afectará al valor original de las variables utilizadas en la llamada. Es como si trabajaramos con una copia, no con el original. No es posible enviar por valor arrays, deberemos hacerlo por referencia.  
     Por referencia: lo que hacemos es enviar a la función la dirección de memoria donde se encuentra la variable o dato. Cualquier modificación SI afectará a las variables utilizadas en la llamada. Trabajamos directamente con el original

Ejemplo:

/\* Paso por valor. \*/

#include <stdio.h>

void intercambio(int,int);

main() /\* Intercambio de valores \*/

{

int a=1,b=2;

printf("a=%d y b=%d",a,b);

intercambio(a,b); /\* llamada \*/

printf("a=%d y b=%d",a,b);

}

void intercambio (int x,int y)

{

int aux;

aux=x;

x=y;

y=aux;

printf("a=%d y b=%d",x,y);

}

Para enviar un valor por referencia se utiliza el símbolo & ( ampersand ) delante de la variable enviada. Esto le indica al compilador que la función que se ejecutará tendra que obtener la dirección de memoria en que se encuentra la variable.  
     Vamos a fijarnos en los ejemplos. En el ejemplo anterior podrás comprobar que antes y después de la llamada, las variables mantienen su valor. Solamente se modifica en la función intercambio ( paso por valor ).  
     En el siguiente ejemplo podrás ver como las variables intercambian su valor tras la llamada de la función ( paso por referencia ).  
     Las variables con un \* son conocidas como punteros, el único dato en 'C' que puede almacenar una dirección de memoria.

Ejemplo:

/\* Paso por referencia. \*/

#include <stdio.h>

void intercambio(int \*,int \*);

main() /\* Intercambio de valores \*/

{

int a=1,b=2;

printf("a=%d y b=%d",a,b);

intercambio(&a,&b); /\* llamada \*/

printf("a=%d y b=%d",a,b);

}

void intercambio (int \*x,int \*y)

{

int aux;

aux=\*x;

\*x=\*y;

\*y=aux;

printf("a=%d y b=%d",\*x,\*y);

}

.- Los argumentos de la función main  
  
     Ya hemos visto que las funciones pueden recibir argumentos. Pues bién, la función main no podía ser menos y también puede recibir argumentos, en este caso desde el exterior.  
     Los argumentos que puede recibir son:  
     argc: es un contador. Su valor es igual al número de argumentos escritos en la línea de comandos, contando el nombre del programa que es el primer argumento.  
     argv: es un puntero a un array de cadenas de caracteres que contiene los argumentos, uno por cadena.  
     En este ejemplo vamos a ver un pequeño programa que escribirá un saludo por pantalla. El programa FUNCION6.EXE.

ejemplo:

/\* Argumentos de la main. \*/

#include <stdio.h>

main(int argc,char \*argv[]) /\* argumentos \*/

{

printf("\nCurso de Programación en C - Copyright (c) 1997-2001, Sergio Pacho\n");

printf("Programa de ejemplo.\n\n");

if (argc<2)

{

printf("Teclee: funcion6 su\_nombre");

exit(1); /\* fin \*/

}

printf("Hola %s",argv[1]);

}

**ARRAYS**

Un array es un identificador que referencia un conjunto de datos del mismo tipo. Imagina un tipo de dato int; podremos crear un conjunto de datos de ese tipo y utilizar uno u otro con sólo cambiar el índice que lo referencia. El índice será un valor entero y positivo. En C los arrays comienzan por la posición 0.  
  
**Vectores**  
     Un vector es un array unidimensional, es decir, sólo utiliza un índice para referenciar a cada uno de los elementos. Su declaración será:

tipo nombre [tamaño];

     El tipo puede ser cualquiera de los ya conocidos y el tamaño indica el número de elementos del vector ( se debe indicar entre corchetes [ ] ). En el ejemplo puedes observar que la variable i es utilizada como índice, el primer for sirve para rellenar el vector y el segundo para visualizarlo. Como ves, las posiciones van de 0 a 9 ( total 10 elementos ).

Ejemplo:

/\* Declaración de un array. \*/

#include <stdio.h>

main() /\* Rellenamos del 0 - 9 \*/

{

int vector[10],i;

for (i=0;i<10;i++) vector[i]=i;

for (i=0;i<10;i++) printf(" %d",vector[i]);

}

Podemos inicializar (asignarle valores) un vector en el momento de declararlo. Si lo hacemos así no es necesario indicar el tamaño. Su sintaxis es:

tipo nombre []={ valor 1, valor 2...}

     Ejemplos:

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

char vector[]="programador";

char vector[]={'p','r','o','g','r','a','m','a','d','o','r'};

     Una particularidad con los vectores de tipo char (cadena de caracteres), es que deberemos indicar en que elemento se encuentra el fin de la cadena mediante el caracter nulo (\0). Esto no lo controla el compilador, y tendremos que ser nosotros los que insertemos este caracter al final de la cadena.  
     Por tanto, en un vector de 10 elementos de tipo char podremos rellenar un máximo de 9, es decir, hasta vector[8]. Si sólo rellenamos los 5 primeros, hasta vector[4], debemos asignar el caracter nulo a vector[5]. Es muy sencillo: vector[5]='\0'; .  
     Ahora veremos un ejemplo de como se rellena un vector de tipo char.

Ejemplo:

/\* Vector de tipo char. \*/

#include <stdio.h>

main() /\* Rellenamos un vector char \*/

{

char cadena[20];

int i;

for (i=0;i<19 && cadena[i-1]!=13;i++)

cadena[i]=getche( );

if (i==19) cadena[i]='\0';

else cadena[i-1]='\0';

printf("\n%s",cadena);

}

 Podemos ver que en el for se encuentran dos condiciones:  
          1.- Que no se hayan rellenado todos los elementos (i<19).  
          2.- Que el usuario no haya pulsado la tecla ENTER, cuyo código ASCII es 13. (cadena[x-i]!=13).  
     También podemos observar una nueva función llamada getche( ), que se encuentra en conio.h. Esta función permite la entrada de un caracter por teclado. Después se encuentra un if, que comprueba si se ha rellenado todo el vector. Si es cierto, coloca el caracter nulo en el elemento nº20 (cadena[19]). En caso contrario tenemos el else, que asigna el caracter nulo al elemento que almacenó el caracter ENTER.  
     En resumen: al declarar una cadena deberemos reservar una posición más que la longitud que queremos que tenga dicha cadena.  
  
     .- Llamadas a funciones con arrays  
  
     Como ya se comentó en el tema anterior, los arrays únicamente pueden ser enviados a una función por referencia. Para ello deberemos enviar la dirección de memoria del primer elemento del array. Por tanto, el argumento de la función deberá ser un puntero.

Ejemplo:

/\* Envío de un array a una función. \*/

#include <stdio.h>

void visualizar(int []); /\* prototipo \*/

main() /\* rellenamos y visualizamos \*/

{

int array[25],i;

for (i=0;i<25;i++)

{

printf("Elemento nº %d",i+1);

scanf("%d",&array[i]);

}

visualizar(&array[0]);

}

void visualizar(int array[]) /\* desarrollo \*/

{

int i;

for (i=0;i<25;i++) printf("%d",array[i]);

}

En el ejemplo se puede apreciar la forma de enviar un array por referencia. La función se podía haber declarado de otra manera, aunque funciona exactamente igual:

declaración o prototipo

void visualizar(int \*);

desarrollo de la función

void visualizar(int \*array)

**Matrices**  
     Una matriz es un array multidimensional. Se definen igual que los vectores excepto que se requiere un índice por cada dimensión.  
     Su sintaxis es la siguiente:

tipo nombre [tamaño 1][tamaño 2]...;

     Una matriz bidimensional se podría representar gráficamente como una tabla con filas y columnas.  
     La matriz tridimensional se utiliza, por ejemplo, para trabajos gráficos con objetos 3D.  
     En el ejemplo puedes ver como se rellena y visualiza una matriz bidimensional. Se necesitan dos bucles para cada una de las operaciones. Un bucle controla las filas y otro las columnas.

Ejemplo:

/\* Matriz bidimensional. \*/

#include <stdio.h>

main() /\* Rellenamos una matriz \*/

{

int x,i,numeros[3][4];

/\* rellenamos la matriz \*/

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

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

scanf("%d",&numeros[x][i]);

/\* visualizamos la matriz \*/

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

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

printf("%d",numeros[x][i]);

}

Si al declarar una matriz también queremos inicializarla, habrá que tener encuenta el orden en el que los valores son asignados a los elementos de la matriz. Veamos algunos ejemplos:

int numeros[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

     quedarían asignados de la siguiente manera:

numeros[0][0]=1 numeros[0][1]=2 numeros[0][2]=3 numeros[0][3]=4

numeros[1][0]=5 numeros[1][1]=6 numeros[1][2]=7 numeros[1][3]=8

numeros[2][0]=9 numeros[2][1]=10 numeros[2][2]=11 numeros[2][3]=12

     También se pueden inicializar cadenas de texto:

char dias[7][10]={"lunes","martes","miércoles","jueves","viernes","sábado","domingo"};

     Para referirnos a cada palabra bastaría con el primer índice:

printf("%s",dias[i]);

**PUNTEROS**

Un puntero es una variable que contiene la dirección de memoria de otra variable. Se utilizan para pasar información entre una función y sus puntos de llamada.  
  
 **Declaración**  
  
     Su sintaxis es la siguiente:

tipo \*nombre;

     Donde nombre es, naturalmente, el nombre de la variable, y tipo es el tipo del elemento cuya dirección almacena el puntero.  
  
 **Operadores**  
  
     Existen dos operadores especiales para trabajar con punteros: & y \*.  
     El primero devuelve la dirección de memoria de su operando. Por ejemplo, si queremos guardar en el puntero x la dirección de memoria de la variable num, deberemos hacer lo siguiente:

x=&num;

     El segundo devuelve el valor de la variable cuya dirección es contenida por el puntero. Este ejemplo sitúa el contenido de la variable apuntada por x, es decir num, en la variable a:

a=\*x;

**Asignación**  
  
     Los punteros se asignan igual que el resto de las variables. El programa ejemplo mostrará las direcciones contenidas en p1 y p2, que será la misma en ambos punteros.

Ejemplo:

/\* Asignaciones de punteros. \*/

#include <stdio.h>

main() /\* Asignamos direcciones \*/

{

int a;

int \*p1,\*p2;

p1=&a;

p2=p1;

printf("%p %p",p1,p2);

}

**Aritmética de direcciones**  
  
     Es posible desplazar un puntero recorriendo posiciones de memoria. Para ello podemos usar los operadores de suma, resta, incremento y decremento (+, -, ++, - -). Si tenemos un puntero ( p1 ) de tipo int ( 2 bytes ), apuntando a la posición 30000 y hacemos: p1=p1+5; el puntero almacenará la posición 30010, porque apunta 5 enteros por encima ( 10 bytes más ).

**ESTRUCTURAS**

**Concepto de estructura**  
Una estructura es un conjunto de una o más variables, de distinto tipo, agrupadas bajo un mismo nombre para que su manejo sea más sencillo.  
Su utilización más habitual es para la programación de bases de datos, ya que están especialmente indicadas para el trabajo con registros o fichas.  
    La sintaxis de su declaración es la siguiente:

struct tipo\_estructura

{

tipo\_variable nombre\_variable1;

tipo\_variable nombre\_variable2;

tipo\_variable nombre\_variable3;

};

     Donde tipo\_estructura es el nombre del nuevo tipo de dato que hemos creado. Por último, tipo\_variable y nombre\_variable son las variables que forman parte de la estructura.  
     Para definir variables del tipo que acabamos de crear lo podemos hacer de varias maneras, aunque las dos más utilizadas son éstas:

Una forma de definir la estructura:

struct trabajador

{

char nombre[20];

char apellidos[40];

int edad;

char puesto[10];

};

struct trabajador fijo, temporal;

Otra forma:

struct trabajador

{

char nombre[20];

char apellidos[40];

int edad;

char puesto[10];

}fijo, temporal;

     En el primer caso declaramos la estructura, y en el momento en que necesitamos las variables, las declaramos. En el segundo las declaramos al mismo tiempo que la estructura. El problema del segundo método es que no podremos declarar más variables de este tipo a lo largo del programa. Para poder declarar una variable de tipo estructura, la estructura tiene que estar declarada previamente. Se debe declarar antes de la función main.  
     El manejo de las estructuras es muy sencillo, así como el acceso a los campos ( o variables ) de estas estructuras. La forma de acceder a estos campos es la siguiente:

variable.campo;

     Donde variable es el nombre de la variable de tipo estructura que hemos creado, y campo es el nombre de la variable que forma parte de la estructura. Lo veremos mejor con un ejemplo basado en la estructura definida anteriormente:

temporal.edad=25;

     Lo que estamos haciendo es almacenar el valor 25 en el campo edad de la variable temporal de tipo trabajador.  
     Otra característica interesante de las estructuras es que permiten pasar el contenido de una estructura a otra, siempre que sean del mismo tipo naturalmente:

fijo=temporal;

     Al igual que con los otros tipos de datos, también es posible inicializar variables de tipo estructura en el momento de su declaración:

     struct trabajador fijo={"Pedro","Hernández Suárez", 32, "gerente"};

     Si uno de los campos de la estructura es un array de números, los valores de la inicialización deberán ir entre llaves:

struct notas

{

char nombre[30];

int notas[5];

};

struct notas alumno={"Carlos Pérez",{8,7,9,6,10}};

**Estructuras y funciones**  
  
     Podemos enviar una estructura a una función de las dos maneras conocidas:  
     1.- Por valor: su declaración sería:

void visualizar(struct trabajador);

     Después declararíamos la variable fijo y su llamada sería:

visualizar(fijo);

     Por último, el desarrollo de la función sería:

void visualizar(struct trabajador datos)

Ejemplo:

/\* Paso de una estructura por valor. \*/

#include <stdio.h>

struct trabajador

{

char nombre[20];

char apellidos[40];

int edad;

char puesto[10];

};

void visualizar(struct trabajador);

main() /\* Rellenar y visualizar \*/

{

struct trabajador fijo;

printf("Nombre: ");

scanf("%s",fijo.nombre);

printf("\nApellidos: ");

scanf("%s",fijo.apellidos);

printf("\nEdad: ");

scanf("%d",&fijo.edad);

printf("\nPuesto: ");

scanf("%s",fijo.puesto);

visualizar(fijo);

}

void visualizar(struct trabajador datos)

{

printf("Nombre: %s",datos.nombre);

printf("\nApellidos: %s",datos.apellidos);

printf("\nEdad: %d",datos.edad);

printf("\nPuesto: %s",datos.puesto);

}

2.- Por referencia: su declaración sería:

void visualizar(struct trabajador \*);

     Después declararemos la variable fijo y su llamada será:

visualizar(&fijo);

     Por último, el desarrollo de la función será:

void visualizar(struct trabajador \*datos)

     Fíjate que en la función visualizar, el acceso a los campos de la variable datos se realiza mediante el operador ->, ya que tratamos con un puntero. En estos casos siempre utilizaremos el operador ->. Se consigue con el signo menos seguido de mayor que.

Ejemplo:

/\* Paso de una estructura por referencia. \*/

#include <stdio.h>

struct trabajador

{

char nombre[20];

char apellidos[40];

int edad;

char puesto[10];

};

void visualizar(struct trabajador \*);

main() /\* Rellenar y visualizar \*/

{

struct trabajador fijo;

printf("Nombre: ");

scanf("%s",fijo.nombre);

printf("\nApellidos: ");

scanf("%s",fijo.apellidos);

printf("\nEdad: ");

scanf("%d",&fijo.edad);

printf("\nPuesto: ");

scanf("%s",fijo.puesto);

visualizar(&fijo);

}

void visualizar(struct trabajador \*datos)

{

printf("Nombre: %s",datos->nombre);

printf("\nApellidos: %s",datos->apellidos);

printf("\nEdad: %d",datos->edad);

printf("\nPuesto: %s",datos->puesto);

}

**Arrays de estructuras**  
     Es posible agrupar un conjunto de elementos de tipo estructura en un array. Esto se conoce como array de estructuras:

struct trabajador

{

char nombre[20];

char apellidos[40];

int edad;

};

struct trabajador fijo[20];

     Así podremos almacenar los datos de 20 trabajadores. Ejemplos sobre como acceder a los campos y sus elementos: para ver el nombre del cuarto trabajador, fijo[3].nombre;. Para ver la tercera letra del nombre del cuarto trabajador, fijo[3].nombre[2];. Para inicializar la variable en el momento de declararla lo haremos de esta manera:

struct trabajador fijo[20]={{"José","Herrero Martínez",29},{"Luis","García Sánchez",46}};

**Typedef**  
     El lenguaje 'C' dispone de una declaración llamada typedef que permite la creación de nuevos tipos de datos. Ejemplos:

typedef int entero; /\* acabamos de crear un tipo de dato llamado entero \*/

entero a, b=3; /\* declaramos dos variables de este tipo \*/

     Su empleo con estructuras está especialmente indicado. Se puede hacer de varias formas:

Una forma de hacerlo:

struct trabajador

{

char nombre[20];

char apellidos[40];

int edad;

};

typedef struct trabajador datos;

datos fijo,temporal;

Otra forma:

typedef struct

{

char nombre[20];

char apellidos[40];

int edad;

}datos;

datos fijo,temporal;

**FICHEROS**

Ahora veremos la forma de almacenar datos que podremos recuperar cuando deseemos. Estudiaremos los distintos modos en que podemos abrir un fichero, así como las funciones para leer y escribir en él.

**Apertura**  
     Antes de abrir un fichero necesitamos declarar un puntero de tipo FILE, con el que trabajaremos durante todo el proceso. Para abrir el fichero utilizaremos la función fopen( ).  
  
     Su sintaxis es:

FILE \*puntero;

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

     donde puntero es la variable de tipo FILE, nombre del fichero es el nombre que daremos al fichero que queremos crear o abrir. Este nombre debe ir encerrado entre comillas. También podemos especificar la ruta donde se encuentra o utilizar un array que contenga el nombre del archivo ( en este caso no se pondrán las comillas ). Algunos ejemplos:

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

w crea un fichero de escritura. Si ya existe lo crea de nuevo.

w+ crea un fichero de lectura y escritura. Si ya existe lo crea de nuevo.

a abre o crea un fichero para añadir datos al final del mismo.

a+ abre o crea un fichero para leer y añadir datos al final del mismo.

r abre un fichero de lectura.

r+ abre un fichero de lectura y escritura.

     Modo binario

wb crea un fichero de escritura. Si ya existe lo crea de nuevo.

w+b crea un fichero de lectura y escritura. Si ya existe lo crea de nuevo.

ab abre o crea un fichero para añadir datos al final del mismo.

a+b abre o crea un fichero para leer y añadir datos al final del mismo.

rb abre un fichero de lectura.

r+b abre un fichero de lectura y escritura.

     La función fopen devuelve, como ya hemos visto, un puntero de tipo FILE. Si al intentar abrir el fichero se produjese un error ( por ejemplo si no existe y lo estamos abriendo en modo lectura ), la función fopen devolvería NULL. Por esta razón es mejor controlar las posibles causas de error a la hora de programar. Un ejemplo:

FILE \*pf;

pf=fopen("datos.txt","r");

if (pf == NULL) printf("Error al abrir el fichero");

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

freopen(nombre del fichero,"modo de apertura",puntero);

     donde nombre del fichero es el nombre del nuevo fichero que queremos abrir, luego el modo de apertura, y finalmente el puntero que va a ser reasignado.

**Cierre**  
  
     Una vez que hemos acabado nuestro trabajo con un fichero es recomendable cerrarlo. Los ficheros se cierran al finalizar el programa pero el número de estos que pueden estar abiertos es limitado. Para cerrar los ficheros utilizaremos la función fclose( );.  
  
     Esta función cierra el fichero, cuyo puntero le indicamos como parámetro. Si el fichero se cierra con éxito devuelve 0.

fclose(puntero);

     Un ejemplo ilustrativo aunque de poca utilidad:

FILE \*pf;

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

if ( pf == NULL ) printf ("Error al abrir el fichero");

else fclose(pf);

**Escritura y lectura**  
     A continuación veremos las funciones que se podrán utilizar dependiendo del dato que queramos escribir y/o leer en el fichero.  
  
  
     Un caracter

fputc( variable\_caracter , puntero\_fichero );

     Escribimos un caracter en un fichero ( abierto en modo escritura ). Un ejemplo:

FILE \*pf;

char letra='a';

if (!(pf=fopen("datos.txt","w"))) /\* otra forma de controlar si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else fputc(letra,pf);

fclose(pf);

fgetc( puntero\_fichero );

     Lee un caracter de un fichero ( abierto en modo lectura ). Deberemos guardarlo en una variable. Un ejemplo:

FILE \*pf;

char letra;

if (!(pf=fopen("datos.txt","r"))) /\* controlamos si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else

{

letra=fgetc(pf);

printf("%c",letra);

fclose(pf);

}

     Un número entero

putw( variable\_entera, puntero\_fichero );

     Escribe un número entero en formato binario en el fichero. Ejemplo:

FILE \*pf;

int num=3;

if (!(pf=fopen("datos.txt","wb"))) /\* controlamos si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else

{

fputw(num,pf); /\* también podíamos haber hecho directamente: fputw(3,pf); \*/

fclose(pf);

}

getw( puntero\_fichero );

     Lee un número entero de un fichero, avanzando dos bytes después de cada lectura. Un ejemplo:

FILE \*pf;

int num;

if (!(pf=fopen("datos.txt","rb"))) /\* controlamos si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else

{

num=getw(pf);

printf("%d",num);

fclose(pf);

}

     Una cadena de caracteres

fputs( variable\_array, puntero\_fichero );

     Escribe una cadena de caracteres en el fichero. Ejemplo:

FILE \*pf;

char cad="Me llamo Vicente";

if (!(pf=fopen("datos.txt","w"))) /\* controlamos si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else

{

fputs(cad,pf); /\* o también así: fputs("Me llamo Vicente",pf); \*/

fclose(pf);

}

fgets( variable\_array, variable\_entera, puntero\_fichero );

     Lee una cadena de caracteres del fichero y la almacena en variable\_array. La variable\_entera indica la longitud máxima de caracteres que puede leer. Un ejemplo:

FILE \*pf;

char cad[80];

if (!(pf=fopen("datos.txt","rb"))) /\* controlamos si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else

{

fgets(cad,80,pf);

printf("%s",cad);

fclose(pf);

}

     Con formato

fprintf( puntero\_fichero, formato, argumentos);

     Funciona igual que un printf pero guarda la salida en un fichero. Ejemplo:

FILE \*pf;

char nombre[20]="Santiago";

int edad=34;

if (!(pf=fopen("datos.txt","w"))) /\* controlamos si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else

{

fprintf(pf,"%20s%2d\n",nombre,edad);

fclose(pf);

}

fscanf( puntero\_fichero, formato, argumentos );

     Lee los argumentos del fichero. Al igual que con un scanf, deberemos indicar la dirección de memoria de los argumentos con el símbolo& ( ampersand ). Un ejemplo:

FILE \*pf;

char nombre[20];

int edad;

if (!(pf=fopen("datos.txt","rb"))) /\* controlamos si se produce un error \*/

{

printf("Error al abrir el fichero");

exit(0); /\* abandonamos el programa \*/

}

else

{

fscanf(pf,"%20s%2d\",nombre,&edad);

printf("Nombre: %s Edad: %d",nombre,edad);

fclose(pf);

}

     Estructuras

fwrite( \*buffer, tamaño, nº de veces, puntero\_fichero );

     Se utiliza para escribir bloques de texto o de datos, estructuras, en un fichero. En esta función, \*buffer será la dirección de memoria de la cuál se recogerán los datos; tamaño, el tamaño en bytes que ocupan esos datos y nº de veces, será el número de elementos del tamaño indicado que se escribirán.

fread( \*buffer, tamaño, nº de veces, puntero\_fichero );

     Se utiliza para leer bloques de texto o de datos de un fichero. En esta función, \*buffer es la dirección de memoria en la que se almacenan los datos; tamaño, el tamaño en bytes que ocupan esos datos y nº de veces, será el número de elementos del tamaño indicado que se leerán.  
     Puedes encontrar ejemplos sobre la apertura y cierre de ficheros, así como de la lectura y escritura de datos, en el archivo IMAGECAT.C. Se trata de un programa que crea un catálogo en formato HTML a partir de las imágenes que se encuentran en un directorio determinado.  
  
     Otras funciones para ficheros

rewind( puntero\_fichero );

     Sitúa el puntero al principio del archivo.

fseek( puntero\_fichero, long posicion, int origen );

     Sitúa el puntero en la posicion que le indiquemos. Como origen podremos poner:

0 o SEEK\_SET, el principio del fichero

1 o SEEK\_CUR, la posición actual

2 o SEEK\_END, el final del fichero

rename( nombre1, nombre2 );

     Su función es exactamente la misma que la que conocemos en MS-DOS. Cambia el nombre del fichero nombre1 por un nuevo nombre,nombre2.   
remove( nombre );

     Como la función del DOS del, podremos eliminar el archivo indicado en nombre.   
  
     Detección de final de fichero

feof( puntero\_fichero );

     Siempre deberemos controlar si hemos llegado al final de fichero cuando estemos leyendo, de lo contrario podrían producirse errores de lectura no deseados. Para este fin disponemos de la función feof( ). Esta función retorna 0 si no ha llegado al final, y un valor diferente de 0si lo ha alcanzado.  
     Pues con esto llegamos al final del tema. Espero que no haya sido muy pesado. No es necesario que te aprendas todas las funciones de memoria. Céntrate sobre todo en las funciones fputs( ), fgets( ), fprintf( ), fwrite( ) y fread( ). Con estas cinco se pueden gestionar los ficheros perfectamente.

**GESTION DINAMICA DE MEMORIA**

**Funciones**  
     Como veremos después, la gestión dinámica memoria se realiza mediante estructuras dinámicas de datos. Fíjate que se repite la palabradinámica. Estas estructuras se diferencian de las estáticas ( arrays y estructuras ), en que no tienen un tamaño fijo, es decir, no tenemos que indicar su tamaño al declararlas, sino que podremos aumentarlo o disminuirlo en tiempo de ejecución, cuando se esté ejecutando la aplicación. Como puedes ver, las estructuras dinámicas son de gran utilidad. A continuación veremos las funciones que se encargan de reservar y liberar memoria durante la ejecución, que se encuentran en la librería alloc.h:  
  
     malloc( tamaño );  
  
     Esta función reserva en memoria una zona de tamaño bytes, y devuelve un puntero al inicio de esa zona. Si no hubiera suficiente memoria retornaría NULL. Más adelante veremos algunos ejemplos.  
  
     free( puntero );  
  
     Esta función libera de la memoria la zona que habíamos reservado anteriormente con la función malloc. También podremos ver algún ejemplo en la página siguiente.

**Estructuras dinámicas de datos**  
     En función de la forma en que se relacionan existen varios tipos de estructuras de datos. Este tipo de estructuras son autorreferenciadas, es decir, contienen entre sus campos un puntero de su mismo tipo. Las más utilizadas son:

- pilas

- colas

- listas

     Las pilas  
  
     Este tipo de estructuras se caracteriza porque todas las operaciones se realizan en el mismo lado. Es de tipo LIFO ( Last In First Out ), el último elemento en entrar es el primero en salir.

Ejemplo:

/\* Ejemplo de una pila. \*/

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <alloc.h>

void insertar(void);

void extraer(void);

void visualizar(void);

struct pila

{

char nombre[20];

struct pila \*ant;

}\*CAB=NULL,\*AUX=NULL;

main() /\* Rellenar, extraer y visualizar \*/

{

char opc;

do

{

clrscr(); /\* borramos la pantalla \*/

gotoxy(30,8); /\* columna 30, fila 8 \*/

printf("1.- Insertar");

gotoxy(30,10);

printf("2.- Extraer");

gotoxy(30,12);

printf("3.- Visualizar la pila");

gotoxy(30,14);

printf("4.- Salir");

opc=getch( );

switch(opc)

{

case '1':

insertar( );

break;

case '2':

extraer( );

break;

case '3':

visualizar( );

}

}while (opc!='4');

}

void insertar(void)

{

AUX=(struct pila \*)malloc(sizeof(struct pila));

clrscr();

printf("Nombre: ");

gets(AUX->nombre);

if (CAB==NULL)

{

CAB=AUX;

AUX->ant=NULL;

}

else

{

AUX->ant=CAB;

CAB=AUX;

}

}

void extraer(void)

{

if (CAB==NULL) return;

AUX=CAB;

CAB=CAB->ant;

free(AUX);

}

void visualizar(void)

{

if (CAB==NULL) return;

clrscr();

AUX=CAB;

while (AUX!=NULL)

{

printf("Nombre: %s\n",AUX->nombre);

AUX=AUX->ant;

}

getch( );

}

La estructura tipo que utilizaremos será ésta:

struct pila

{

tipo variables;

struct pila \*ant;

}\*CAB=NULL,\*AUX=NULL;

     donde tipo variables serán las diferentes variables que guardaremos en la estructura, struct pila \*ant es un puntero que apunta al elemento de tipo pila introducido anteriormente, \*CAB será donde guardaremos el último elemento insertado en la pila y \*AUX nos servirá para guardar elementos temporalmente y para recorrer la pila al visualizarla.  
     Antes de insertar un elemento, deberemos comprobar si la pila está vacía o no. Si lo estuviera deberemos insertar el primer elemento:

CAB=AUX;

CAB->ant=NULL;

     Si ya hubiera algún elemento crearemos uno nuevo apuntado por AUX y haremos que AUX->ant apunte a CAB, que en este momento contiene la dirección del elemento insertado anteriormente. Tras esto haremos que CAB apunte al último elemento insertado, que será la nueva cabeza de la pila:

AUX->ant=CAB;

CAB=AUX;

     Para extraer un elemento de la pila deberemos hacer que AUX apunte a la misma dirección que CAB, después haremos que CAB apunte a CAB->ant, con lo que el elemento anterior pasará a ser la cabeza de la pila. Tras esto, solo queda liberar la memoria de la zona apuntada por AUX. No olvides controlar si existe algún elemento ( si CAB es igual a NULL la pila está vacía ):

if (CAB==NULL) return;

AUX=CAB;

CAB=CAB->ant;

free(AUX);

     Por último, para visualizar los elementos de la pila, haremos que el puntero auxiliar AUX apunte a la cabeza de la pila, o sea, a CAB. Tras esto iremos visualizando el contenido de la pila, haciendo que AUX tome la dirección de AUX->ant, mientras AUX sea distinto deNULL. También es importante controlar que la pila no esté vacía.

if (CAB==NULL) return;

AUX=CAB;

while (AUX!=NULL)

{

printf("%s",AUX->nombre);

AUX=AUX->ant;

};

Estructura gráfica de una pila:

Las colas  
  
     Este tipo de estructuras se caracteriza porque insertamos los elementos por un lado y los extraemos por el otro lado. Es de tipo FIFO (First In First Out ), el primer elemento en entrar es el primero en salir. Para gestionar la cola utilizaremos 3 punteros ( para la pila solo eran necesarios 2 ).

Ejemplo:

/\* Ejemplo de una cola. \*/

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <alloc.h>

void insertar(void);

void extraer(void);

void visualizar(void);

struct cola

{

char nombre[20];

struct cola \*sig;

}\*CAB=NULL,\*AUX=NULL,\*FIN=NULL;

main() /\* Rellenar, extraer y visualizar \*/

{

char opc;

do

{

clrscr();

gotoxy(30,8);

printf("1.- Insertar");

gotoxy(30,10);

printf("2.- Extraer");

gotoxy(30,12);

printf("3.- Visualizar la cola");

gotoxy(30,14);

printf("4.- Salir");

opc=getch( );

switch(opc)

{

case '1':

insertar( );

break;

case '2':

extraer( );

break;

case '3':

visualizar( );

}

}while (opc!='4');

}

void insertar(void)

{

AUX=(struct cola \*)malloc(sizeof(struct cola));

clrscr();

printf("Nombre: ");

gets(AUX->nombre);

AUX->sig=NULL;

if (FIN==NULL)

FIN=CAB=AUX;

else

{

FIN->sig=AUX;

FIN=AUX;

}

}

void extraer(void)

{

if (CAB==NULL) return;

AUX=CAB;

CAB=CAB->sig;

free(AUX);

}

void visualizar(void)

{

if (CAB==NULL) return;

clrscr();

AUX=CAB;

while (AUX!=NULL)

{

printf("Nombre: %s\n",AUX->nombre);

AUX=AUX->sig;

}

getch();

}

 La estructura que utilizaremos será:

struct cola

{

tipo variables;

struct cola \*sig;

}\*CAB=NULL,\*AUX=NULL,\*FIN=NULL;

     donde tipo variables serán las diferentes variables que guardaremos en la estructura, struct cola \*sig es un puntero que apunta al elemento de tipo cola introducido a continuación, \*CAB será donde guardaremos el primer elemento insertado en la cola, \*AUX nos servirá para guardar elementos temporalmente y para recorrer la cola al visualizarla y \*FIN tomará la dirección del último elemento insertado.  
     Antes de insertar un elemento, deberemos comprobar si la cola está vacía o no. Si lo está deberemos insertar el primer elemento:

if (FIN==NULL)

CAB=FIN=AUX;

     Si ya existiera algún elemento haremos que FIN->sig apunte al elemento de AUX y a continuación haremos que FIN tome la dirección deAUX, con lo que FIN apuntará al último elemento insertado.

FIN->sig=AUX;

FIN=AUX;

     Para extraer un elemento de la cola haremos que el puntero auxiliar AUX tome la dirección del primer elemento insertado, que hemos guardado en CAB. Tras esto haremos que CAB apunte a CAB->sig, es decir, que tome la dirección del segundo elemento insertado, que ahora pasará a ser el primero. Luego liberaremos la zona de memoria apuntada por AUX:

AUX=CAB; /\* Deberemos controlar que no esté vacía: if (CAB==NULL) return; \*/

CAB=CAB->sig;

free(AUX);

     Para visualizar la cola comprobaremos que existan elementos, esto es, que FIN sea distinto de NULL. Hecho esto asignaremos a AUXla dirección de CAB e iremos recorriendo la cola hasta que AUX sea igual a NULL.

AUX=CAB; /\* Deberemos controlar que no esté vacía: if (CAB==NULL) return; \*/

while(AUX!=NULL)

{

printf("%s",AUX->nombre);

AUX=AUX->sig;

}

Estructura gráfica de una cola:

Las listas  
  
     Este tipo de estructuras se caracteriza porque los elementos están enlazados entre sí, de manera que además de las acciones habituales de insertar, extraer y visualizar también podremos buscar un elemento. Para gestionar la lista utilizaremos 4 punteros.

Ejemplo:

/\* Ejemplo de una lista. \*/

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <alloc.h>

void insertar(void);

void extraer(void);

void visualizar(void);

struct lista

{

int num;

struct lista \*sig;

}\*CAB=NULL,\*AUX=NULL,\*F=NULL,\*P=NULL;

main() /\* Rellenar, extraer y visualizar \*/

{

char opc;

do

{

clrscr( );

gotoxy(30,8);

printf("1.- Insertar");

gotoxy(30,10);

printf("2.- Extraer");

gotoxy(30,12);

printf("3.- Visualizar la lista");

gotoxy(30,14);

printf("4.- Salir");

opc=getch( );

switch(opc)

{

case '1':

insertar( );

break;

case '2':

extraer( );

break;

case '3':

visualizar( );

}

}while (opc!='4');

}

/\* A continuación insertaremos el elemento que

vamos a crear en la posición que le corresponda,

teniendo en cuenta que la lista deberá quedar

ordenada de menor a mayor. El puntero P comprueba

si el campo num de un elemento es menor que el

campo num del elemento introducido. El puntero

F se quedará apuntando al elemento de la posición

anterior al elemento que hemos insertado \*/

void insertar(void)

{

AUX=(struct lista \*)malloc(sizeof(struct lista));

clrscr( );

printf("Introduce un número: ");

scanf("%d",&AUX->num);

AUX->sig=NULL;

if (CAB==NULL)

CAB=AUX;

else if (CAB->num > AUX->num)

{

AUX->sig=CAB;

CAB=AUX;

}

else

{

P=F=CAB;

while (P->num < AUX->num && P!=NULL)

{

if (P==CAB) P=P->sig;

else

{

P=P->sig;

F=F->sig;

}

}

AUX->sig=F->sig;

F->sig=AUX;

}

}

void extraer(void)

{

int var;

if (CAB==NULL) return;

clrscr( );

printf("Introduce el número a extraer: ");

scanf("%d",&var);

if (CAB->num==var)

{

P=CAB;

CAB=CAB->sig;

free(P);

}

else

{

P=F=CAB;

while (P->num != var && P!=NULL)

{

if (P==CAB) P=P->sig;

else

{

P=P->sig;

F=F->sig;

}

}

if (P==NULL) return;

F->sig=P->sig;

free(P);

}

}

void visualizar(void)

{

if (CAB==NULL) return;

clrscr( );

AUX=CAB;

while (AUX!=NULL)

{

printf("Número: %d\n",AUX->num);

AUX=AUX->sig;

}

getch( );

}

La estructura que utilizaremos será:

struct lista

{

tipo variables;

struct lista \*sig;

}\*CAB=NULL,\*AUX=NULL,\*F=NULL,\*P=NULL;

     donde tipo variables serán las variables que guardaremos en la estructura, struct lista \*sig es un puntero que apunta al elemento de tipo lista introducido a continuación, \*CAB será donde guardaremos el primer elemento de la lista, \*AUX nos servirá para guardar elementos temporalmente y para recorrer la lista al visualizarla, \*P para comparar los valores introducidos y ordenarlos, y \*F, que apuntará al elemento anterior al último introducido.  
     Antes de insertar un elemento, deberemos comprobar si la lista está vacía o no. Si lo está deberemos insertar el primer elemento:

if (CAB==NULL) CAB=AUX;

     Si ya existiera algún elemento haremos que P y F apunten al primero de la lista. Si el elemento introducido fuera menor que el primero de la lista, haríamos que el nuevo elemento pasara a ser el primero, y el que hasta ahora era el primero, pasaría a ser el segundo.

if (AUX->num < CAB->num){

AUX->sig=CAB;

CAB=AUX;

}

     Para extraer un elemento de la lista solicitaremos un número, si el número introducido se corresponde con el campo num de uno de los elementos, éste será extraído de la lista. Deberemos controlar que la lista no esté vacía y que el elemento con el número solicitado exista.  
     Fíjate en el ejemplo, en la función extraer. Si CAB es igual a NULL, será que la lista está vacía, y si P es igual a NULL al salir del whilesignificará que no se ha encontrado ningún elemento que contenga el número introducido.  
     Para visualizar la lista comprobaremos que existan elementos, es decir, que CAB sea distinto de NULL. Hecho esto asignaremos a AUXla dirección de CAB e iremos recorriendo la lista mientras AUX sea distinto de NULL.

if (CAB==NULL) return;

AUX=CAB;

while(AUX!=NULL)

{

printf("%d",AUX->num);

AUX=AUX->sig;

}

  Aquí finaliza el tema de la gestión dinámica de memoria. Es un tema algo complejo hasta que se asimila el concepto y funcionamiento de las diferentes estructuras, pero tras conseguirlo ya no tiene ningún secreto. Si alguna vez no recuerdas su funcionamiento siempre es una buena solución coger papel y lápiz, dibujar una pila, cola o lista gráficamente y simular la introducción de elementos, escribiendo la situación de los punteros en cada momento.  
     Existen otras estructuras, como las listas doblemente enlazadas. La única diferencia con la lista que conocemos es que en las primeras cada elemento guarda la dirección del anterior y del posterior. Sería una estructura como esta:

struct lista\_doble

{

char nombre[20];

struct lista\_doble \*ant;

struct lista\_doble \*sig;

};

Su funcionamiento es muy similar al de una lista normal. Puedes intentar hacerla tu mismo.  
Otras estructuras, como los árboles son más complejas y menos utilizadas.

**PROGRAMACION GRAFICA**

**Conceptos básicos**  
  
     El estándar de 'C' no define ninguna función gráfica debido a las grandes diferencias entre las interfaces de los distintos tipos de hardware. Nosotros veremos el conjunto de funciones que utiliza Turbo C. La resolución más habitual del modo gráfico en Turbo C es de640x480x16.  
  
     Inicialización del modo gráfico  
  
     Para poder trabajar en modo gráfico primero deberemos inicializarlo. Las funciones a utilizar son estas.

detectgraph (int \*tarjeta , int \*modo);

     Detecta el tipo de tarjeta que tenemos instalado. Si en el primer argumento retorna -2 indica que no tenemos ninguna tarjeta gráfica instalada (cosa bastante improbable).  
initgraph (int \*tarjeta , int \*modo , "path");

     Inicializa el modo gráfico ( primero hay que usar detectgraph ). En path deberemos indicar el directorio donde se encuentra el archivoEGAVGA.BGI.  
int graphresult( );

     Retorna el estado del modo gráfico. Si no se produce ningún error devuelve 0, de lo contrario devuelve un valor entre -1 y -16.  
char grapherrormsg(int error);

     Retorna un puntero al mensaje de error indicado por graphresult.  
  
     Finalización del modo gráfico

closegraph( );

     Cierra el modo gráfico y nos devuelve al modo texto.  
restorecrtmode( );

     Reestablece el modo de video original ( anterior a initgraph ).

Ejemplo:

/\* Inicialización del modo gráfico. \*/

#include <graphics.h>

main() /\* Inicializa y finaliza el modo gráfico. \*/

{

int tarjeta, modo, error;

detectgraph(&tarjeta,&modo);

initgraph(&tarjeta,&modo,"C:\\TC\\BGI");

error=graphresult( );

if (error)

{

printf("%s",grapherrormsg(error));

}

else

{

getch( );

closegraph( );

}

}

**Funciones**

int getmaxx( );

     Retorna la coordenada máxima horizontal, probablemente 639. Ej: hm=getmaxx( );

int getmaxy( );

     Retorna la coordenada máxima vertical, probablemente 479. Ej: vm=getmaxy( );

int getx( );

     Retorna la coordenada actual horizontal. Ej: hact=getx( );

int gety( );

     Retorna la coordenada actual vertical. Ej: vact=gety( );

moveto(int x , int y);

     Se mueve a las coordenadas indicadas. Ej: moveto(320,240);

setcolor(color);

     Selecciona el color de dibujo y texto indicado. Ej: setcolor(1); o setcolor(BLUE);

setbkcolor(color);

     Selecciona el color de fondo indicado. Ej: setbkcolor(4); o setbkcolor(RED);

int getcolor( );

     Retorna el color de dibujo y texto actual. Ej: coloract=getcolor( );

int getbkcolor( );

     Retorna el color de fondo actual. Ej: fondoact=getbkcolor( );

int getpixel(int x , int y);

     Retorna el color del pixel en x,y. Ej: colorp=getpixel(120,375);

cleardevice( );

    Borra la pantalla. Ej: cleardevice( );  
  
     Funciones de dibujo

putpixel(int x , int y , color);

     Pinta un pixel en las coordenadas y color indicados. Ej: putpixel(100,50,9);

line(int x1 , int y1 , int x2 , int y2);

     Dibuja una linea desde x1,y1 a x2,y2. Ej: line(20,10,150,100);

circle(int x , int y , int radio);

     Dibuja un círculo del radio indicado y con centro en x,y. Ej: circle(320,200,20);

rectangle(int x1 , int y1 , int x2 , int y2);

     Dibuja un rectángulo con la esquina superior izquierda en x1,y1 y la inferior derecha en x2,y2. Ej: rectangle(280,210,360,270);

arc(int x , int y, int angulo1 , int angulo2 , int radio);

     Dibuja un arco cuyo centro está en x,y, de radio r, y que va desde angulo1 a angulo2. Ej: arc(200,200,90,180,40);

setlinestyle(int estilo, 1 , grosor);

     Selecciona el estilo de linea a utilizar. El estilo puede tomar un valor de 0 a 4. El grosor puede tomar dos valores: 1 = normal y 3 = ancho. Ej: setlinestyle(2,1,3);  
  
     Funciones de relleno

floodfill(int x , int y , int frontera);

     Rellena el area delimitada por el color indicado en frontera comenzando desde x,y. Ej: floodfill(100,30,12);

setfillstyle(int pattern , int color);

     Selecciona el patrón y el color de relleno. El patrón puede tomar un valor de 0 a 12 Ej: setfillstyle(1,9);

bar(int x1 , int y1, int x2 , int y2);

     Dibuja una barra ( rectángulo ) y si es posible la rellena. Ej: bar(200,200,400,300);

bar3d(int x1 , int y1, int x2 , int y2 , int profundidad , int tapa);

     Dibuja una barra en 3d, son los mismos valores que bar además de la profundidad y la tapa: 0 si la queremos sin tapa y 1 si la queremos con tapa. Ej: bar3d(100,100,400,150,40,1);

pieslice(int x , int y , int angulo1 , int angulo2 , int radio);

     Dibuja un sector. Hace lo mismo que arc, pero además lo cierra y lo rellena. Ej: pieslice(250,140,270,320,50);  
  
     Funciones de escritura de texto

outtextxy(int x , int y , char \*);

     Muestra el texto indicado ( puede ser un array o puede escribirse al llamar a la función ) en las coordenadas x,y. Ej:outtextxy(50,50,"Esto es texto en modo gráfico");

settextstyle(int fuente , int dirección , int tamaño);

     Selecciona el estilo del texto. Las fuentes más comunes son las que van de 0 a 4. La dirección puede ser: 0 = horizontal y 1 = vertical. El tamaño puede tomar un valor de 1 a 10. Ej: settextstyle(2,0,5);

setviewport(int x1 , int y1 , int x2 , int y2 , int tipo);

     Define una porción de pantalla para trabajar con ella. La esquina superior izquierda está determinada por x1,y1 y la inferior derecha por x2,y2. Para tipo podemos indicar 1, en cuyo caso no mostrará la parte de un dibujo que sobrepase los límites del viewport, o distinto de 1, que sí mostrará todo el dibujo aunque sobrepase los límites. Al activar un viewport, la esquina superior izquierda pasará a tener las coordenadas (0,0). Para volver a trabajar con la pantalla completa, deberemos escribir: viewport(0,0,639,479,1);.

clearviewport( );

     Borra el contenido del viewport.  
  
     Aquí concluye el tema del modo gráfico. Hay algunas funciones más, aunque su complejidad es mayor. Generalmente no se suelen utilizar más que las aquí descritas, pero puedes investigar en la ayuda de Turbo C para conocer alguna otra.

**APENDICE**

En este capítulo y para finalizar veremos los ficheros de cabecera, donde están declaradas las funciones que utilizaremos habitualmente.  
  
**Librería stdio.h**

     printf  
     Función: Escribe en la salida estándar con formato.  
  
     Sintaxis: printf(formato , arg1 , ...);  
  
    scanf  
     Función: Lee de la salida estándar con formato.  
     Sintaxis: scanf(formato , arg1 , ...);  
  
     puts  
     Función: Escribe una cadena y salto de linea.  
     Sintaxis: puts(cadena);  
  
     gets  
     Función: Lee y guarda una cadena introducida por teclado.  
     Sintaxis: gets(cadena);  
  
     fopen  
     Función: Abre un fichero en el modo indicado.  
     Sintaxis: pf=fopen(fichero , modo);  
  
     fclose  
     Función: Cierra un fichero cuyo puntero le indicamos.  
     Sintaxis: fclose(pf);  
  
     fprintf  
     Función: Escribe con formato en un fichero.  
     Sintaxis: fprintf(pf , formato , arg1 , ...);  
  
     fgets  
     Función: Lee una cadena de un fichero.  
     Sintaxis: fgets(cadena , longitud , pf);

**Librería stdlib.h**  
  
     atof  
     Función: Convierte una cadena de texto en un valor de tipo float.  
     Sintaxis: numflo=atof(cadena);  
  
     atoi  
     Función: Convierte una cadena de texto en un valor de tipo entero.  
     Sintaxis: nument=atoi(cadena);  
  
     itoa  
     Función: Convierte un valor numérico entero en una cadena de texto. La base generalmente será 10, aunque se puede indicar otra distinta.  
     Sintaxis: itoa(número , cadena , base);  
  
     exit  
     Función: Termina la ejecución y abandona el programa.  
     Sintaxis: exit(estado); /\* Normalmente el estado será 0 \*/

**Librería conio.h**  
  
     clrscr  
     Función: Borra la pantalla.  
     Sintaxis: clrscr( );  
  
     clreol  
     Función: Borra desde la posición del cursor hasta el final de la linea.  
     Sintaxis: clreol( );  
  
     gotoxy  
     Función: Cambia la posición del cursor a las coordenadas indicadas.  
     Sintaxis: gotoxy(columna , fila);  
  
     textcolor  
     Función: Selecciona el color de texto (0 - 15).  
     Sintaxis: textcolor(color);  
  
     textbackground  
     Función: Selecciona el color de fondo (0 - 7).  
     Sintaxis: textbackground(color);  
  
    wherex  
     Función: Retorna la columna en la que se encuentra el cursor.  
     Sintaxis: col=wherex( );  
  
     wherey  
     Función: Retorna la fila en la que se encuentra el cursor.  
     Sintaxis: fila=wherey( );  
  
     getch  
     Función: Lee y retorna un único caracter introducido mediante el teclado por el usuario. No muestra el caracter por la pantalla.  
     Sintaxis: letra=getch( );  
  
     getche  
     Función: Lee y retorna un único caracter introducido mediante el teclado por el usuario. Muestra el caracter por la pantalla.  
     Sintaxis: letra=getche( );

**Librería string.h**  
  
     strlen  
     Función: Calcula la longitud de una cadena.  
     Sintaxis: longitud=strlen(cadena);  
  
     strcpy  
     Función: Copia el contenido de una cadena sobre otra.  
     Sintaxis: strcpy(copia , original);  
  
     strcat  
     Función: Concatena dos cadenas.  
     Sintaxis: strcat(cadena1 , cadena2);  
  
     strcmp  
     Función: Compara el contenido de dos cadenas. Si cadena1 < cadena2 retorna un número negativo. Si cadena1 > cadena2, un n£mero positivo, y si cadena1 es igual que cadena2 retorna 0 ( o NULL ).  
     Sintaxis: valor=strcmp(cadena1 , cadena2);

**Librería graphics.h**  
     Además de las que vimos al estudiar la programación gráfica existen otras funciones. Aquí tienes algunas de ellas.  
  
     getmaxcolor  
     Función: Retorna el valor más alto de color disponible.  
     Sintaxis: mcolor=getmaxcolor( );  
  
     setactivepage  
     Función: En modos de video con varias páginas, selecciona la que recibirá todas las operaciones y dibujos que realicemos.  
     Sintaxis: setactivepage(página); /\* En modo VGA página = 0 ó 1 \*/  
  
     setvisualpage  
     Función: En modos de video con varias páginas, selecciona la que se visualizará por pantalla.  
     Sintaxis: setvisualpage(página);

**Librería dir.h**  
  
     En esta librería encontraremos una serie de rutinas que nos permitirán realizar operaciones básicas con directorios y unidades de disco.  
  
     chdir  
     Función: Cambia el directorio actual.  
     Sintaxis: chdir(ruta); /\* Podemos indicar la unidad: chdir("a:\\DATOS"); \*/  
  
     getcwd  
     Función: Lee del sistema el nombre del directorio de trabajo.  
     Sintaxis: getcwd(directorio,tamañocad) /\* Lee el directorio y la unidad \*/  
  
     getdisk  
     Función: Lee del sistema la unidad actual.  
     Sintaxis: disk=getdisk( ) + 'A'; /\* Retorna un entero: 0 = A: , 1 = B: ... \*/  
  
     mkdir  
     Función: Crea un directorio.  
     Sintaxis: mkdir(nombre);

**Funciones interesantes**  
  
     **fflush(stdin)**  
     Función: Limpia el buffer de teclado.  
     Sintaxis: fflush(stdin);  
     Prototipo: stdio.h  
  
     sizeof  
     Función: Operador que retorna el tamaño en bytes de una variable.  
     Sintaxis: tamaño=sizeof(variable);  
  
     cprintf  
**Función:** Funciona como el printf pero escribe en el color que hayamos activado con la función textcolor sobre el color activado con textbackground.  
     Sintaxis: cprintf(formato , arg1 , ...);  
     Prototipo: conio.h  
     kbhit  
     Función: Espera la pulsación de una tecla para continuar la ejecución.  
     Sintaxis: while (!kbhit( )) /\* Mientras no pulsemos una tecla... \*/  
     Prototipo: conio.h  
  
     **RANDOM**  
     Función: Retorna un valor aleatorio entre 0 y num-1.  
     Sintaxis: valor=random(num); /\* También necesitamos la función randomize \*/  
     Prototipo: stdlib.h  
  
**RANDOMIZE**  
     Función: Inicializa el generador de números aleatorios. Deberemos llamarlo al inicio de la función en que utilicemos el random. También deberemos utilizar el include time.h, ya que randomize hace una llamada a la función time, incluída en este último archivo.  
     Sintaxis: randomize( );  
     Prototipo: stdio.h  
  
     **SYSTEM**  
     Función: Ejecuta el comando indicado. Esto incluye tanto los comandos del sistema operativo, como cualquier programa que nosotros le indiquemos. Al acabar la ejecución del comando, volverá a la linea de código situada a continuación de la sentencia system.  
     Sintaxis: system(comando); /\* p.ej: system("arj a programa"); \*/  
     Prototipo: stdlib.h

**¿Qué es JavaScript?**

[JavaScript](http://www.instantweb.com/~foldoc/foldoc.cgi?query=javascript)® es el lenguaje interpretado orientado a objetos desarrollado por Netscape que se utiliza en millones de páginas web y aplicaciones de servidor en todo el mundo. JavaScript de Netscape es un superconjunto del lenguaje de scripts estándar de la edición de ECMA-262 3 ([ECMAScript](https://developer.mozilla.org/en/ECMAScript)) que presenta sólo leves diferencias respecto a la norma publicada.

Contrariamente a la falsa idea popular, JavaScript no es "Java interpretativo". En pocas palabras, JavaScript es un lenguaje de programación dinámico que soporta construcción de objetos [basado en prototipos](https://developer.mozilla.org/en/JavaScript/Guide/Details_of_the_Object_Model). La sintaxis básica es similar a Java y C++ con la intención de reducir el número de nuevos conceptos necesarios para aprender el lenguaje. Las construcciones del lenguaje, tales como sentencias if, y bucles for y while, y bloques switch y try ... catch funcionan de la misma manera que en estos lenguajes (o casi).

JavaScript puede funcionar como [lenguaje procedimental](http://www.instantweb.com/~foldoc/foldoc.cgi?query=procedural&action=Search) y como [lenguaje orientado a objetos](http://www.instantweb.com/~foldoc/foldoc.cgi?query=object+oriented&action=Search). Los objetos se crean programáticamente añadiendo métodos y propiedades a lo que de otra forma serían objetos vacíos en tiempo de ejecución, en contraposición a las definiciones sintácticas de clases comunes en los lenguajes compilados como C++ y Java. Una vez se ha construido un objeto, puede usarse como modelo (o prototipo) para crear objetos similares.

Las capacidades dinámicas de JavaScript incluyen construcción de objetos en tiempo de ejecución, listas variables de parámetros, variables que pueden contener funciones, creación de scripts dinámicos (mediante [eval](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/eval)), introspección de objetos (mediante for ... in), y recuperación de código fuente (los programas de JavaScript pueden decompilar el cuerpo de funciones a su código fuente original).

Los objectos intrínsecos son Number, String, Boolean, Date, RegExp y Math.

Para una discusión más detallada de la programación en JavaScript sigue los enlaces a[recursos sobre JavaScript](https://developer.mozilla.org/es/docs/Web/JavaScript/Acerca_de_JavaScript) que hay a continuación.

**¿Qué implementaciones de JavaScript están disponibles?**

mozilla.org alberga dos implementaciones de JavaScript. La primera implementación de JavaScript fue creada por Brendan Eich en Netscape, y desde entonces ha sido actualizada (en JavaScript 1.5) para cumplir con ECMA-262 Edición 5. Este motor, cuyo nombre en código es [SpiderMonkey](https://developer.mozilla.org/en/SpiderMonkey), se implementa en C. El motor [Rhino](https://developer.mozilla.org/en/Rhino), creado principalmente por Norris Boyd (también en Netscape) es una implementación de Javascript en Java. Al igual que SpiderMonkey, Rhino cumple con ECMA-262 Edición 3.

Varias optimizaciones tales como TraceMonkey (Firefox 3.5), JägerMonkey (Firefox 4) e IonMonkey fueron agregadas al motor de JavaScript de SpiderMonkey durante el tiempo.

Además de las implementaciones anteriores, existen otros motores de JavaScript populares tales como:

[V8](http://code.google.com/p/v8/) de Google, el cual es usado en el navegador Google Chrome.

[JavaScriptCore](http://www.webkit.org/projects/javascript/index.html) (SquirrelFish/Nitro) usado en algunos navegadores basados en WebKit tales como Apple Safari.

[Carakan](http://my.opera.com/ODIN/blog/carakan-faq) en Opera.

El motor [Chakra](http://es.wikipedia.org/wiki/Chakra_%28int%C3%A9rprete_de_JScript%29), usado en Internet Explorer, es tecnicamente un motor de JScript, en lugar de un motor de JavaScript.

Cada motor de JavaScript de mozilla.org expone una API pública que las aplicaciones pueden llamar para aprovechar el soporte de JavaScript. Por el momento, el entorno de host más común de JavaScript son los navegadores web porque suelen utilizar la API pública para crear "objetos de host" responsables de reflejar el [DOM](http://www.w3.org/DOM/) en JavaScript.

Otra aplicación común de JavaScript es como lenguaje interpretado de lado del servidor (web). Un servidor web escrito en JavaScript podría exponer objetos host que representen objetos de una petición y una respuesta HTTP, los cuales podrían ser manipulados por un programa en JavaScript para generar páginas web de manera dinámica.

Para más información sobre cómo incrustar JavaScript en tus propias aplicaciones, sigue cualquiera de los enlaces de SpiderMonkey o de Rhino que te mostramos a continuación, o visítanos en el grupo de noticias netscape.public.mozilla.jseng.

**Recursos sobre JavaScript**

[**SpiderMonkey**](https://developer.mozilla.org/en/SpiderMonkey)

Información específica para la incrustación del motor JavaScript en C (también conocido como SpiderMonkey).

[**Rhino**](https://developer.mozilla.org/en/Rhino)

Información específica para la implementación de JavaScript escrita en Java (también conocida como Rhino).

[**Recursos del lenguaje**](https://developer.mozilla.org/en/JavaScript/Language_Resources)

Punteros a los estándares JavaScript publicados, documentos LiveConnect y el trabajo sobre JavaScript 2.0.

[**Tamarin Project**](https://developer.mozilla.org/en/Tamarin)

Información sobre la máquina virtual de código abierto y alto rendimiento para Javascript 2.0.

### JavaScript en el lado servidor

Netscape introdujo una implementación de [script del lado del servidor](https://es.wikipedia.org/wiki/Script_del_lado_del_servidor) con [Netscape Enterprise Server](https://es.wikipedia.org/w/index.php?title=Netscape_Enterprise_Server&action=edit&redlink=1), lanzada en diciembre de 1994 (poco después del lanzamiento de JavaScript para navegadores web).[14](https://es.wikipedia.org/wiki/JavaScript) [15](https://es.wikipedia.org/wiki/JavaScript) A partir de mediados de la década de los 2000, ha habido una proliferación de implementaciones de JavaScript para el lado servidor.[Node.js](https://es.wikipedia.org/wiki/Node.js) es uno de los notables ejemplos de JavaScript en el lado del servidor, siendo usado en proyectos importantes.[16](https://es.wikipedia.org/wiki/JavaScript) [17](https://es.wikipedia.org/wiki/JavaScript)

### Desarrollos posteriores

JavaScript se ha convertido en uno de los lenguajes de programación más populares en internet. Al principio, sin embargo, muchos desarrolladores renegaban del lenguaje porque el público al que va dirigido lo formaban publicadores de artículos y demás aficionados, entre otras razones.[18](https://es.wikipedia.org/wiki/JavaScript) La llegada de [Ajax](https://es.wikipedia.org/wiki/AJAX) devolvió JavaScript a la fama y atrajo la atención de muchos otros programadores. Como resultado de esto hubo una proliferación de un conjunto de [frameworks](https://es.wikipedia.org/wiki/Framework_para_aplicaciones_web) y [librerías](https://es.wikipedia.org/wiki/Biblioteca_(inform%C3%A1tica)) de ámbito general, mejorando las prácticas de programación con JavaScript, y aumentado el uso de JavaScript fuera de los [navegadores web](https://es.wikipedia.org/wiki/Navegador_web), como se ha visto con la proliferación de entornos [JavaScript del lado del servidor](https://es.wikipedia.org/w/index.php?title=Server-side_JavaScript&action=edit&redlink=1). En enero de 2009, el proyecto [CommonJS](https://es.wikipedia.org/w/index.php?title=CommonJS&action=edit&redlink=1) fue inaugurado con el objetivo de especificar una librería para uso de tareas comunes principalmente para el desarrollo fuera del navegador web.[19](https://es.wikipedia.org/wiki/JavaScript)

En junio de 2015 se cerró y publicó el estándar [ECMAScript](https://es.wikipedia.org/wiki/ECMAScript) 6[20](https://es.wikipedia.org/wiki/JavaScript) [21](https://es.wikipedia.org/wiki/JavaScript) con un soporte irregular entre navegadores[22](https://es.wikipedia.org/wiki/JavaScript) y que dota a JavaScript de características avanzadas que se echaban de menos y que son de uso habitual en otros lenguajes como, por ejemplo, módulos para organización del código, verdaderas [clases](https://es.wikipedia.org/wiki/Clase_(inform%C3%A1tica)) para [programación orientada a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos), expresiones de flecha, iteradores, generadores o promesas para programación asíncrona.

La versión 7 de ECMAScript se conoce como ECMAScript 2016[23](https://es.wikipedia.org/wiki/JavaScript) , y es la última versión disponible, publicada en junio de 2016. Se trata de la primera versión para la que se usa un nuevo procedimiento de publicación anual y un proceso de desarrollo abierto[24](https://es.wikipedia.org/wiki/JavaScript) .

**jQuery**

**La librería jQuery**

**¿Qué es jQuery?**

Podemos empezar diciendo que jQuery es una libreria o "framework" de javascript. Sin embargo para el que no conozca estos términos empezaremos por definir lo que es una libreria en programación.

Una librería o "framework" consiste en uno o varios archivos de código de programación (javascript en este caso) que facilitan la tarea del programador. La función de una librería consiste en liberar al programador de repetir código para tareas habituales, para ello normalmente acorta el código que tiene que escribir, y crea una serie de objetos, funciones y métodos que dan gran parte del trabajo ya hecho.

En nuestro caso jQuery consiste en un archivo que recopila las principales tareas que suelen hacerse con javascript y que al incorporarlo a la página al programador le resulta más fácil crear el código. No solo libera de tareas repetitivas al programador, sino que también simplifica la manera de trabajar al tener que escribir menos código.

jQuery es uno de los frameworks más utilizados. Su simplicidad y su efectividad son la gran ventaja que tiene jQuery. Si queremos dedicarnos a la programación web debemos conocer varias librerías javascript, tal vez jQuery y MooTools sean las mas usadas, jQuery por su facilidad en el manejo, y MooTools por ser una de las más completas.

**LA LIBRERÍA MOOTOOLS**

**¿Qué es MooTools?**

MooTools es una librería o "framework" de javascript. Las librerías o "framework" consisten en uno o varios archivos de código de programación (javascript en este caso) que facilitan la tarea del programador. Se utiliza un código más sencillo o más corto para realizar las tareas habituales o más comunes.

La librería MooTools consiste en varios archivos. Hay un archivo central o núcleo, también llamado "Core", y varios archivos o plugins que utilizaremos o no dependiendo de lo que queramos hacer.

## Ventajas

## Como otras tantas bibliotecas JavaScript, MooTools aporta al usuario muchas ventajas. Alguna de ellas:

* Es un Framework modular y extendible, el desarrollador puede elegir (específicamente) que componentes usar y cuales no.
* MooTools es orientado a objetos y sigue los [principios DRY](https://es.wikipedia.org/wiki/No_te_repitas), que hacen de él un Framework rico, potente y eficiente.
* Componente avanzado de efectos (Effects), con transiciones (Transitions), de función parabólica, optimizadas y utilizadas por multitud de desarrolladores Flash.
* Framework desarrollado por programadores para programadores.

Es una herramienta que será usada en wordlskill

**BOOTSTRAP**

Es un [framework](https://es.wikipedia.org/wiki/Framework) o conjunto de herramientas de [Código abierto](https://es.wikipedia.org/wiki/C%C3%B3digo_abierto) para diseño de sitios y aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en[HTML](https://es.wikipedia.org/wiki/HTML) y [CSS](https://es.wikipedia.org/wiki/Hojas_de_estilo_en_cascada), así como, extensiones de [JavaScript](https://es.wikipedia.org/wiki/JavaScript) opcionales adicionales.

**Características**

Bootstrap tiene un soporte relativamente incompleto para [HTML5](https://es.wikipedia.org/wiki/HTML5) y [CSS 3](https://es.wikipedia.org/wiki/CSS_3), pero es compatible con la mayoría de los navegadores web. La información básica de compatibilidad de sitios web o aplicaciones está disponible para todos los dispositivos y navegadores. Existe un concepto de compatibilidad parcial que hace disponible la información básica de un sitio web para todos los dispositivos y navegadores. Por ejemplo, las propiedades introducidas en CSS3 para las esquinas redondeadas, gradientes y sombras son usadas por Bootstrap a pesar de la falta de soporte de navegadores antiguos. Esto extiende la funcionalidad de la herramienta, pero no es requerida para su uso.

Desde la versión 2.0 también soporta [diseños sensibles](https://es.wikipedia.org/w/index.php?title=Responsive_Web_Design&action=edit&redlink=1). Esto significa que el diseño gráfico de la página se ajusta dinámicamente, tomando en cuenta las características del dispositivo usado (Computadoras, tabletas, teléfonos móviles).

Bootstrap es de código abierto y está disponible en GitHub. Los desarrolladores están motivados a participar en el proyecto y a hacer sus propias contribuciones a la plataforma.

**SQL**

SQL (por sus siglas en [inglés](https://es.wikipedia.org/wiki/Idioma_ingl%C3%A9s) Structured Query Language; en [español](https://es.wikipedia.org/wiki/Idioma_espa%C3%B1ol) lenguaje de consulta estructurada) es un [lenguaje específico del dominio](https://es.wikipedia.org/wiki/Lenguaje_espec%C3%ADfico_del_dominio) que da acceso a un [sistema de gestión de bases de datos relacionales](https://es.wikipedia.org/wiki/Sistema_de_gesti%C3%B3n_de_bases_de_datos_relacionales) que permite especificar diversos tipos de operaciones en ellos. Una de sus características es el manejo del [álgebra](https://es.wikipedia.org/wiki/%C3%81lgebra_relacional) y el [cálculo relacional](https://es.wikipedia.org/wiki/C%C3%A1lculo_relacional) que permiten efectuar[consultas](https://es.wikipedia.org/w/index.php?title=Consulta_(base_de_datos)&action=edit&redlink=1) con el fin de recuperar, de forma sencilla, [información](https://es.wikipedia.org/wiki/Informaci%C3%B3n) de bases de datos, así como hacer cambios en ellas. SQL (Lenguaje estructurado de consulta) es un lenguaje específico de dominio que se usa en la programación y está diseñado para manejar los datos que hay en un sistema de base de datos, o para el procesamiento de flujos de datos en un sistema de gestión de flujos.

Originalmente está basado en el álgebra relacional y en el cálculo relacional, SQL consiste en un lenguaje de definición de datos, un lenguaje de manipulación de datos y un lenguaje de control de datos. El alcance de SQL incluye la inserción de datos, consultas, actualizaciones y borrado, la creación y modificación de esquemas y el control de acceso a los datos. También el SQL a veces se describe como un lenguaje declarativo, también incluye elementos procesales.

SQL fue uno de los primeros lenguajes comerciales para el modelo relacional de [Edgar F.](https://es.wikipedia.org/wiki/Edgar_Frank_Codd) como se describió en su papel de 1970 “ El modelo relacional de datos para grandes bancos de datos compartidos”. A pesar de no adherirse totalmente al modelo relacional descrito por Codd, pasó a ser el lenguaje de base de datos más usado.

SQL pasó a ser el estándar del [Instituto Americano Nacional de estándares](https://es.wikipedia.org/wiki/Instituto_Nacional_Estadounidense_de_Est%C3%A1ndares) en 1986 y de la Organización Internacional para estandarizar en 1987. Desde entonces. El estándar ha sido revisado para incluir más características. A pesar de la existencia de ambos estándares, La mayoría de los códigos SQL no son completamente portables entre sistemas de bases de datos diferentes sin ajustes.

**VISUAL BASIC**

Visual Basic (VB) es un [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n) [dirigido por eventos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_dirigida_por_eventos), desarrollado por [Alan Cooper](https://es.wikipedia.org/wiki/Alan_Cooper) para [Microsoft](https://es.wikipedia.org/wiki/Microsoft). Este lenguaje de programación es un dialecto de [BASIC](https://es.wikipedia.org/wiki/BASIC), con importantes agregados. Su primera versión fue presentada en [1991](https://es.wikipedia.org/wiki/1991), con la intención de simplificar la programación utilizando un [ambiente de desarrollo](https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado) que facilitó en cierta medida la [programación](https://es.wikipedia.org/wiki/Programaci%C3%B3n) misma.

La última versión fue la 6, liberada en 1998, para la que Microsoft extendió el soporte hasta marzo de 2008.

En 2001 Microsoft propuso abandonar el desarrollo basado en la [API](https://es.wikipedia.org/wiki/API) [Win32](https://es.wikipedia.org/wiki/Win32) y pasar a un [framework](https://es.wikipedia.org/wiki/Framework) o marco común de librerías, independiente de la versión del sistema operativo, [.NET Framework](https://es.wikipedia.org/wiki/.NET_Framework), a través de [Visual Basic .NET](https://es.wikipedia.org/wiki/Visual_Basic_.NET) (y otros lenguajes como [C Sharp](https://es.wikipedia.org/wiki/C_Sharp) (C#) de fácil transición de código entre ellos); fue el sucesor de Visual Basic 6.

Aunque Visual Basic es de propósito general, también provee facilidades para el desarrollo de aplicaciones de [bases de datos](https://es.wikipedia.org/wiki/Bases_de_datos)usando [Data Access Objects](https://es.wikipedia.org/wiki/Data_Access_Objects), [Remote Data Objects](https://es.wikipedia.org/wiki/Remote_Data_Objects) o [ActiveX Data Objects](https://es.wikipedia.org/wiki/ActiveX_Data_Objects).

Visual Basic contiene un entorno de desarrollo integrado o [IDE](https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado) que integra [editor de textos](https://es.wikipedia.org/wiki/Editor_de_textos) para edición del código fuente, un[depurador](https://es.wikipedia.org/wiki/Depurador), un [compilador](https://es.wikipedia.org/wiki/Compilador) (y [enlazador](https://es.wikipedia.org/wiki/Enlazador)) y un editor de interfaces gráficas o [GUI](https://es.wikipedia.org/wiki/GUI).

Visual Basic dio un paso más en innovación y ahora es posible programar aplicaciones Nativas para Android o IPhone utilizando un software de la compañía Anywhere Software que transforma código Visual Basic (creado por dicha compañía) en APPs 100 % nativas en Java para ambos sistemas operativos de dispositivos móviles.[1](https://es.wikipedia.org/wiki/Visual_Basic)

# CONCLUSIONES PERSONALES