# LIBRO

***Nombre de la materia: Programación Básica***

***Nombre de la Licenciatura: Informática Administrativa***

***Cuatrimestre: Quinto***

## *Periodo Enero- Abril.*

# Marco Estratégico de Referencia

## Antecedentes históricos

Nuestra Universidad tiene sus antecedentes de formación en el año de 1979 con el inicio de actividades de la normal de educadoras "Edgar Robledo Santiago", que en su momento marcó un nuevo rumbo para la educación de Comitán y del estado de Chiapas. Nuestra escuela fue fundada por el Profesor Manuel Albores Salazar con la idea de traer educación a Comitán, ya que esto representaba una forma de apoyar a muchas familias de la región para que siguieran estudiando.

En el año 1984 inicia actividades el CBTiS Moctezuma Ilhuicamina, que fue el primer bachillerato tecnológico particular del estado de Chiapas, manteniendo con esto la visión en grande de traer educación a nuestro municipio, esta institución fue creada para que la gente que trabajaba por la mañana tuviera la opción de estudiar por las tardes.

La Maestra Martha Ruth Alcázar Mellanes es la madre de los tres integrantes de la familia Albores Alcázar que se fueron integrando poco a poco a la escuela formada por su padre, el Profesor Manuel Albores Salazar; Víctor Manuel Albores Alcázar en julio de 1996 como chofer de transporte escolar, Karla Fabiola Albores Alcázar se integró en la docencia en 1998, Martha Patricia Albores Alcázar en el departamento de cobranza en 1999.

En el año 2002, Víctor Manuel Albores Alcázar formó el Grupo Educativo Albores Alcázar S.C. para darle un nuevo rumbo y sentido empresarial al negocio familiar y en el año 2004 funda la Universidad Del Sureste.

La formación de nuestra Universidad se da principalmente porque en Comitán y en toda la región no existía una verdadera oferta Educativa, por lo que se veía urgente la creación de

una institución de Educación superior, pero que estuviera a la altura de las exigencias de los jóvenes que tenían intención de seguir estudiando o de los profesionistas para seguir preparándose a través de estudios de posgrado.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y educativos de los diferentes campus, así como de crear los diferentes planes estratégicos de expansión de la marca.

## Misión

Satisfacer la necesidad de Educación que promueva el espíritu emprendedor, aplicando altos estándares de calidad académica, que propicien el desarrollo de nuestros alumnos, Profesores, colaboradores y la sociedad, a través de la incorporación de tecnologías en el proceso de enseñanza-aprendizaje.

## Visión

Ser la mejor oferta académica en cada región de influencia, y a través de nuestra plataforma virtual tener una cobertura global, con un crecimiento sostenible y las ofertas académicas innovadoras con pertinencia para la sociedad.

## Valores

- Disciplina
- Honestidad
- Equidad
- Libertad

## Escudo

El escudo del Grupo Educativo Albores Alcázar S.C. está constituido por tres líneas curvas que nacen de izquierda a derecha formando los escalones al éxito. En la parte superior está situado un cuadro motivo de la abstracción de la forma de un libro abierto.

## Eslogan

"Mi Universidad"

## ALBORES

Es nuestra mascota, un Jaguar. Su piel es negra y se distingue por ser líder, trabaja en equipo y obtiene lo que desea. El ímpetu, extremo valor y fortaleza son los rasgos que distinguen.

# PROGRAMACIÓN BÁSICA

**Objetivo general de la asignatura:**

Aplicar los conocimientos que permitan una metodología para la solución de problemas, utilizando la computadora.

# INDICE

OBJETIVO GENERAL

Aplicar los conocimientos que permitan una metodología para la solución de problemas utilizando la computadora.

**UNIDAD I** CONCEPTOS BÁSICOS

1.1.- Orígenes del lenguaje "C".

1.2.- Lenguaje de máquina, lenguajes ensambladores.

1.3.- Compilación y enlazado.

1.4.- El entorno integrado del desarrollo (IDE).

1.5.- Estructura de un programa en "C".

**UNIDAD II** TIPOS DE DATOS SIMPLES

2.1.- Tipos de datos.

2.2.- Declaraciones de variables.

2.3.- Clases de almacenamiento.

2.4.- Sentencias de asignación.

2.5.- Definición de constantes.

2.6.- Operadores.

2.7.- Procedimientos definidos de entrada y salida.

**UNIDAD III** FUNCIONES

3.1.- Definición de función.

3.2.- Llamada de una función.

3.3.- Declaración de una función ( FORWAR).

3.4.- Pasos de parámetros de una función.

3.5.- Funciones predefinidas en “C”.

3.6.- Recursividad.

3.7.- Estructuras de control.

3.7.1.- Sentencias condicionales.

3.7.2.- Ciclos y bucles.

3.7.2.- Etiquetas y GOTO.

3.8.- Tipos de datos estructurados.

3.8.1.- Arrays.

3.8.2.- Estructuras.

3.8.3.- Uniones.

3.8.4.- Tipos de enumerados.

**UNIDAD IV** PUNTEROS

4.1.- Definición de punteros.

4.2.- Operación con punteros.

4.3.- Punteros y Arrays.

4.4.- Punteros y funciones.

4.5.- Asignación dinámica de memoria.

4.6.- Entrada, salida y archivo de disco.

4.6.1.- Flujos y archivos.

4.6.2.- E/ S por consola: GETCHE( ) Y PUTCHAR( ).

4.6.3.- E/S por consola con formato PRITF ( ) Y SCANT( ).

4.6.4.- Manejo de archivos.

6.-Actividades de aprendizaje

**Frente al docente:**

- Retroalimentación de conceptos, ejemplos, aplicaciones y ejercicios propuestos.
- Exposición docente, oral, audiovisual.
- Prácticas en el centro de cómputo.
- Independientes:
- Lecturas obligatorias.
- Resolución de problemas.

**Criterios y procedimientos de evaluación y acreditación:**

| | |
|---|---|
| Trabajos escritos | 30% |
| Actividades áulicas | 20% |
| Examen | 50% |
| Total | 100% |
| Escala de calificaciones | 7-10 |
| Mínima aprobatoria | 7 |

# UNIDAD I

# CONCEPTOS BÁSICOS

## 1.1.- Orígenes del lenguaje "C".

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los

Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL. Al igual que B, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

Se trata de un lenguaje débilmente tipificado de medio nivel pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

La primera estandarización del lenguaje C fue en ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido vulgarmente como ANSI C.

Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portátil entre plataformas y/o arquitecturas.

Uno de los objetivos de diseño del lenguaje C es que sólo sean necesarias unas pocas instrucciones en lenguaje máquina para traducir cada elemento del lenguaje, sin que haga falta un soporte intenso en tiempo de ejecución. Es muy posible escribir C a bajo nivel de abstracción; de hecho, C se usó como intermediario entre diferentes lenguajes.

En parte a causa de ser de relativamente bajo nivel y de tener un modesto conjunto de características, se pueden desarrollar compiladores de C fácilmente. En consecuencia, el lenguaje C está disponible en un amplio abanico de plataformas (seguramente más que cualquier otro lenguaje). Además, a pesar de su naturaleza de bajo nivel, el lenguaje se desarrolló para incentivar la programación independiente de la máquina. Un programa escrito cumpliendo los estándares e intentando que sea portátil puede compilarse en muchos computadores.

C se desarrolló originalmente (conjuntamente con el sistema operativo Unix, con el que ha estado asociado mucho tiempo) por programadores para programadores. Sin embargo, ha alcanzado una popularidad enorme, y se ha usado en contextos muy alejados de la programación de sistemas, para la que se diseñó originalmente.

En 1978, Ritchie y Brian Kernighan publicaron la primera edición de El lenguaje de programación C, también conocido como La biblia de C. Este libro fue durante años la

especificación informal del lenguaje. El lenguaje descrito en este libro recibe habitualmente el nombre de "el C de Kernighan y Ritchie" o simplemente "K&R C" (La segunda edición del libro cubre el estándar ANSI C, descrito más abajo.)

Kernighan y Ritchie introdujeron las siguientes características al lenguaje:

- El tipo de datos struct.
- El tipo de datos long int.
- El tipo de datos unsigned int.

Los operadores (=+ y =-) fueron sustituidos por (+= y -=) para eliminar la ambigüedad sintáctica de expresiones como (i=-10), que se podría interpretar bien como (i =- 10) o bien como (i = -10).

El C de Kernighan y Ritchie es el subconjunto más básico del lenguaje que un compilador debe de soportar. Durante muchos años, incluso tras la introducción del ANSI C, fue considerado "el mínimo común denominador" en el que los programadores debían programar cuando deseaban que sus programas fueran transportables, pues no todos los compiladores soportaban completamente ANSI, y el código razonablemente bien escrito en K&R C es también código ANSI C válido.

En estas primeras versiones de C, las únicas funciones que necesitaban ser declaradas si se usaban antes de la definición de la función eran las que retornaban valores no enteros. Es decir, se presuponía que una función que se usaba sin declaración previa (prototipo) devolvería un entero.

### 1.2.- Lenguaje de máquina, lenguajes ensambladores.

Lenguaje de maquina: El lenguaje de maquina es aquel cuyas instrucciones son directamente entendibles por la computadora y no necesitan traducción posterior para que la UCP pueda comprender y ejecutar el programa.

Las instrucciones en lenguaje maquina se expresan en términos de la unidad de memoria más pequeña (bit) = digito binario 0 o 1, en esencia una secuencia de bits que especifican la operación y las celdas de memoria implicadas en una operación

Ejemplo: Instrucciones en lenguaje de máquina:

0010, 0000, 1001, 1001, 10001, 1110.

Como se observa estas instrucciones son fáciles de leer por una computadora y difíciles para un programador y viceversa. Por esta razón se hace difícil escribir programas en código o lenguaje de máquina. Y se requiere otro lenguaje para comunicarse con la computadora pero que se hace más fácil de escribir y de leer por el programador. Para evitar la tediosa tarea de escribir programas en este lenguaje se han diseñado otros programas de programación que facilitan la escritura y posterior ejecución de los programas.

El lenguaje ensamblador es el lenguaje de programación utilizado para escribir programas informáticos de bajo nivel, y constituye la representación más directa del Código máquina específico para cada arquitectura de computadoras legible por un programador. Aun hoy se utiliza en la programación de handler o manipuladores de dispositivos de hardware.

Características:

1. El código escrito en lenguaje ensamblador posee una cierta dificultad de ser entendido directamente por un ser humano ya que su estructura se acerca más bien al lenguaje máquina, es decir, lenguaje de bajo nivel.

2. El lenguaje ensamblador es difícilmente portable, es decir, un código escrito para un Microprocesador, suele necesitar ser modificado, muchas veces en su totalidad para poder ser usado en otra máquina distinta, aun con el mismo Microprocesador, solo pueden ser reutilizados secciones especiales del código programado.

3. Los programas hechos en lenguaje ensamblador, al ser programado directamente sobre Hardware, son generalmente más rápidos y consumen menos recursos del sistema (memoria RAM y ROM). Al programar cuidadosamente en lenguaje ensamblador se pueden crear programas que se ejecutan más rápidamente y ocupan menos espacio que con lenguajes de alto nivel.

4. Con el lenguaje ensamblador se tiene un control muy preciso de las tareas realizadas por un Microprocesador por lo que se pueden crear segmentos de código difíciles de programar en un lenguaje de alto nivel.

5. También se puede controlar el tiempo en que tarda una Rutina en ejecutarse, e impedir que se interrumpa durante su ejecución.

6. El lenguaje ensamblador es un código estructurado y gravitatorio desarrollado sobre un archivo de programación (.ASM), en el cual pueden existir varios programas, macros o rutinas que pueden ser llamados entre sí.

## 1.3.- Compilación y enlazado.

En programación, cuando se desarrolla un programa, en la fase de codificación se llevará a cabo la compilación, que consiste en que el compilador traducirá el código fuente a código máquina, también llamado código objeto, siempre y cuando, el propio compilador no detecte ningún error en dicho código fuente.

En programación, la fase de enlace sirve para unir el código objeto de varios subprogramas por medio de un enlazador. Cuando se desarrolla un programa, estos pueden utilizar subprogramas y, de cada uno de ellos, su código objeto debe ser enlazado (unido) al código objeto del programa que los utilice. Esto se realiza mediante un programa llamado enlazador, montador o linkador en la fase de enlace.

## 1.4.- El entorno integrado del desarrollo (IDE).

Un entorno de desarrollo integrado (IDE) es un sistema de software para el diseño de aplicaciones que combina herramientas del desarrollador comunes en una sola interfaz gráfica de usuario (GUI). Generalmente, un IDE cuenta con las siguientes características:

1. Editor de código fuente: editor de texto que ayuda a escribir el código de software con funciones como el resaltado de la sintaxis con indicaciones visuales, el relleno automático específico del lenguaje y la comprobación de errores a medida que se escribe el código.
2. Automatización de compilación local: herramientas que automatizan tareas sencillas e iterativas como parte de la creación de una compilación local del software para su uso por parte del desarrollador, como la compilación del código fuente de la computadora en un código binario, el empaquetado del código binario y la ejecución de pruebas automatizadas.
3. Depurador: programa que sirve para probar otros programas y mostrar la ubicación de un error en el código original de forma gráfica.

Los IDE permiten que los desarrolladores comiencen a programar aplicaciones nuevas con rapidez, ya que no necesitan establecer ni integrar manualmente varias herramientas como parte del proceso de configuración. Tampoco es necesario que pasen horas aprendiendo a utilizar diferentes herramientas por separado, gracias a que todas están representadas en la misma área de trabajo. Esto resulta muy útil al incorporar desarrolladores nuevos, porque pueden confiar en un IDE para ponerse al día con los flujos de trabajo y las herramientas estándares de un equipo. De hecho, la mayoría de las características de los IDE están diseñadas para ahorrar tiempo, como el relleno inteligente y la generación automatizada del código, lo cual elimina la necesidad de escribir secuencias enteras de caracteres.

Otras funciones comunes del IDE se encargan de ayudar a los desarrolladores a organizar su flujo de trabajo y solucionar problemas. Los IDE analizan el código mientras se escribe, así que las fallas causadas por errores humanos se identifican en tiempo real. Gracias a que hay una sola GUI que representa todas las herramientas, los desarrolladores pueden ejecutar tareas sin tener que pasar de una aplicación a otra. El resaltado de sintaxis también es común en la mayoría de los IDE, y utiliza indicaciones visuales para distinguir la gramática en el editor de texto. Además, algunos IDE incluyen examinadores de objetos yclases, así como diagramas de jerarquía de clases para ciertos lenguajes.

Es posible desarrollar aplicaciones sin IDE, o que los desarrolladores básicamente diseñen su propio IDE integrando varias herramientas de forma manual con los editores de texto ligero como Vim o Emacs. Para algunos desarrolladores, el beneficio de este enfoque radica en el control y la personalización extrema que ofrece. Sin embargo, en un contexto empresarial, el tiempo que se ahorra, la estandarización del entorno y las funciones de automatización de los IDE modernos normalmente superan las demás consideraciones.

Actualmente, la mayoría de los equipos de desarrollo de las empresas optan por un IDE preconfigurado que se adecue mejor a sus casos de uso específicos; por lo que la pregunta no es si conviene adoptar un IDE, sino cuál elegir.

Hay muchos casos de uso comerciales y técnicos distintos para los IDE, lo cual también significa que hay muchas opciones de IDE propietarios y open source en el mercado. En general, las características distintivas más importantes entre los IDE son las siguientes:

1. Cantidad de lenguajes compatibles: algunos IDE son compatibles con un solo lenguaje, así que son mejores para un modelo de programación específico. Por ejemplo, IntelliJ es conocido principalmente como un IDE de Java. Otros IDE admiten una gran variedad de lenguajes de manera conjunta, como el IDE Eclipse, que admite Java, XML, Python, entre otros.
2. Sistemas operativos compatibles: el sistema operativo de un desarrollador determinará qué tipos de IDE son viables (salvo que el IDE esté basado en la nube), y estarán aún más limitados si la aplicación que se desarrolla está diseñada para el usuario final con un sistema operativo específico (como Android o iOS).
3. Características de automatización: si bien la mayoría de los IDE incluyen tres funciones fundamentales (el editor de texto, la automatización de compilación y el depurador), muchos admiten funciones adicionales, como la refactorización, la búsqueda de código y las herramientas de integración e implementación continúas
4. (CI/CD).
5. Impacto en el rendimiento del sistema: es importante considerar el footprint del
6. IDE en la memoria si el desarrollador desea ejecutar otras aplicaciones con uso intensivo de la memoria al mismo tiempo.
7. Complementos y extensiones: algunos IDE incluyen una función para personalizar los flujos de trabajo de forma que se adapten a las necesidades y preferencias del desarrollador.

La creciente popularidad de las aplicaciones diseñadas para teléfonos inteligentes y tabletas influye en casi todos los sectores, así que muchas empresas deben desarrollar aplicaciones móviles, además de las aplicaciones web tradicionales. Uno de los factores clave en el desarrollo de aplicaciones móviles es la selección de la plataforma. Por ejemplo, si se diseña una aplicación nueva para su uso en iOS, Android y una página web, será mejor comenzar con un IDE que brinde soporte en todas las plataformas para varios sistemas operativos.

Los IDE que se ofrecen como software como servicio (SaaS) basado en la nube brindan varios beneficios únicos en comparación con los entornos de desarrollo locales. Por un lado, al igual que con cualquier oferta de SaaS, no es necesario descargar el sistema de software y configurar las dependencias y los entornos locales, lo cual permite que los desarrolladores comiencen a contribuir con los proyectos rápidamente. Esto también ofrece un nivel de estandarización en todos los entornos de los miembros del equipo que permite solucionar el problema típico de que un elemento funcione bien en una computadora y no en otras.

Por otro lado, dado que el entorno de desarrollo se gestiona de forma centralizada, ningún código reside en la computadora de un solo desarrollador, lo cual alivia las preocupaciones de seguridad y propiedad intelectual.

El impacto de los procesos en los equipos locales también es diferente. Los procesos como la ejecución de compilaciones y las pruebas de conjunto de aplicaciones suelen consumir muchos recursos informáticos. Por eso es probable que los desarrolladores no puedan seguir utilizando las estaciones de trabajo mientras se ejecuta un proceso. Un IDE de SaaS puede distribuir las tareas de larga duración sin monopolizar los recursos informáticos de un equipo local. Normalmente, los IDE de nube no suelen depender de ninguna plataforma, lo cual permite su conexión con diferentes proveedores de nube.

## 1.5.- Estructura de un programa en "C".

Todo programa escrito en C consta de una o más funciones, una de las cuales se llama main. El programa siempre comenzará por la ejecución de la función main. Cada función debe contener:

- Una cabecera de la función, que consta del nombre de la función, seguido de una lista opcional de argumentos encerrados con paréntesis.
- Una lista de declaración de argumentos, si se incluyen estos en la cabecera.
- Una sentencia compuesta, que contiene el resto de la función.

Los argumentos son símbolos que representan información que se le pasa a la función desde otra parte del programa. (También se llaman parámetros a los argumentos).

Cada sentencia compuesta se encierra con un par de llaves, {.....}. Las llaves pueden contener combinaciones de sentencias elementales (denominadas sentencias de expresión) y otras sentencias compuestas. Así las sentencias compuestas pueden estar anidadas, una dentro de otra. Cada sentencia de expresión debe acabar en punto y coma

(;).

12

Los comentarios pueden aparecer en cualquier parte del programa, mientras estén situados entre los delimitadores /* ................ */ (por ejemplo: /*esto es un ejemplo*/). Los comentarios son útiles para identificar los elementos principales de un programa o simplemente para orientar a un posible usuario de ese código.

Todo fichero fuente en C sigue la siguiente estructura; para verla más claramente pondremos un ejemplo con sus correspondientes comentarios, que nos vayan explicando cada una de las partes, muchas de las cosas que se vean, no se conocen aún, pero nos servirán para hacernos una idea de cómo se estructura un programa:

Ejemplo:

#include <stdio.h>

#include <conio.h>

/*#include del sistema:Se deben especificar todos los ficheros de cabecera (ficheros con extensión .h) correspondientes a las librerías de funciones utilizadas.

Son librerías implementadas y listas para que nosotros las usemos,con sólo llamar a la función que tenga implementada dentro dicha librería. Por ejemplo: la instrucción printf está incluida dentro de stdio.h, por tanto, cuando decidamos usarla, tendremos que poner en esta sección:

#include <stdio.h> */

#include <lista.h>

/* #include de la aplicación:Ficheros de cabecera creados para el fichero fuente. Se puede decir que estos son los que yo he creado, que son invocados escribiendo su nombre seguido de .h. (Más tarde explicaremos cómo se crean). Si lo pongo entre signos: < > (como lo tenemos en este ejemplo), dicha librería será buscada en el directorio del compilador INCLUDE que contiene las librerías. Si por el contrario pongo: #include "lista.h", entonces, dicha librería será buscada 1° en el directorio actual, y luego, si no está, será buscada en el directorio del compilador INCLUDE*/

extern void salida(void);

/* externvariables globales externas: Variables globales que voy a definir en otros módulos que voy a usar en este módulo. A salida le hemos asignado el tipo de almacenamiento

extern, pues tiene que ser accedida desde otro archivo distinto de aquel en que es definida; por tanto, ha de ser una función externa. */

#define CIERTO 1

#define FALSO 0

/* #define definición de macros y constantes simbólicas.*/

typedef struct {

int dia;

int mes;

int ano;

}FECHA;

/* typedef definición de tipos: me sirve para crearme un tipo distinto de los preestablecidos de partida.*/

int suma (int , int);

/* Declaración de los prototipos de las funciones implementados en este módulo: La declaración de una función le da información al compilador de una función que va a ser utilizada pero que todavía no ha sido implementada. En particular le dice al compilador qué tipo de datos requiere y cuál devuelve la función. En el ejemplo, la función con nombre suma recibe dos números enteros y da como salida otro número entero.*/

extern int a,b,c;

/* Declaración de variables globales de este módulo:

extern declaración de funciones externas a este módulo: Funciones que se utilizan en este módulo y que están implementadas en otro módulo.

static declaración de las funciones internas no visibles para otros módulos: Funciones que se implementan en este módulo y que no pueden ser utilizadas en otros módulos. */

main(){

............

............

}

int suma(int x,int y){

...........

...........

}

/* Implementación de las funciones: Se implementan todas las funciones del módulo incluida la función main().*/

NOTA: En un segundo archivo encontraríamos la definición de la función externa declarada en este primer archivo de encima (extern void salida (void);), esta definición será:

...........

...........

extern void salida(void)

{

```
 printf("¡Hola!");
 return; /* salimos de la función, y volvemos al lugar donde se le llamó sin devolver nada.*/
}
...........
...........
```

## UNIDAD II

## TIPOS DE DATOS SIMPLES

### 2.1.- Tipos de datos.

En programación, un tipo de dato informático o simplemente tipo es un atributo de los datos que indica al ordenador (y/o al programador) sobre la clase de datos que se va a trabajar. Esto incluye imponer restricciones en los datos, como qué valores pueden tomar y qué operaciones se pueden realizar.

Los tipos de datos comunes son: números enteros, números con signo (negativos), números de coma flotante (decimales), cadenas alfanuméricas, estados (booleano), etc.

Algunos tipos de datos usados en C++:

| Data Types | Size in Bytes | Can contain: |
|---|---|---|
| boolean | 1 | true (1) or false (0) |
| char | 1 | ASCII character or signed value between -128 and 127 |
| unsigned char, byte, uint8_t | 1 | ASCII character or unsigned value between 0 and 255 |
| int, short | 2 | signed value between -32,768 and 32,767 |
| unsigned int, word, uint16_t | 2 | unsigned value between 0 and 65,535 |
| long | 4 | signed value between -2,147,483,648 and 2,147,483,647 |
| unsigned long, uint32_t | 4 | unsigned value between 0 and 4,294,967,295 |
| float, double | 4 | floating point value between - 3.4028235E+38 and 3.4028235E+38 (Note that double is the same as a float on this platform.) |

Mi Universidad

Byte:

Byte almacena un valor numérico de 8 bits sin decimales. Tienen un rango entre 0 y 255.

Sin signo.

int (entero):

Enteros son un tipo de datos primarios que almacenan valores numéricos de 16 bits sin decimales comprendidos en el rango 32,767 a -32,768.

Nota: Las variables de tipo entero “int” pueden sobrepasar su valor máximo o mínimo como consecuencia de una operación. Por ejemplo, si x = 32767 y hacemos x++, entonces x pasará a ser -32.768.

Las constantes enteras son números utilizados en el sketch, estos números son tratados como enteros, pero podemos cambiar su comportamiento.

Las constantes enteras son tratadas como base 10 (Decimal), pero con una notación especial podemos cambiar su representación en otras bases.

- Binario – B00001110
- Octal – 0173
- Hexadecimal – 0x7C3
- Para forzar a formato unsigned: 78U ó 78u
- Para forzar a formato long: 1000L ó 1000l
- Para forzar a formato unsigned long: 2521UL ó 2521ul

Long (entero largo):

El formato de variable numérica de tipo extendido “long” se refiere a números enteros (tipo 32 bits = 4 bytes) sin decimales que se encuentran dentro del rango -2147483648 a

2147483647.

Float (decimales):

El formato de dato del tipo “coma flotante” o “float” se aplica a los números con decimales. Los números de coma flotante tienen una mayor resolución que los de 32 bits que ocupa con un rango comprendido 3.4028235E+38 a -3.4028235E+38.

Los números de punto flotante no son exactos, y pueden producir resultados extraños en las comparaciones. Los cálculos matemáticos de coma flotante son también mucho más lentos que los del tipo de números enteros, por lo que debe evitarse su uso si es posible.

En Arduino el tipo de dato double es igual que el float.

Las constantes de coma flotante se usan para facilitar la lectura del código, pero aunque no se use, el compilador no va a dar error y se ejecutará normalmente.

- 10.0 se evalúa como 10
- 2.34E5 ó 67e-12 (expresado en notación científica)

Para entenderlo mejor: la representación de coma flotante (en inglés floating point, 'punto flotante') es una forma de notación científica usada en las CPU, GPU, FPU, etc, con la cual se pueden representar números racionales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas.

El estándar para la representación en coma flotante es el IEEE 754.

- Signo (s) 1: negativo, 0: positivo (bit 31)
- Mantisa (M) La mantisa incluye 23 bits (bit 0.. 22). Representa la parte derecha de número decimal.
- Exponente (e) El exponente incluye 8 bits (bit 23..30).

| $Signo$ | $Exponente$ | $ParteSignificativa$ | |
|---|---|---|---|
| 1 | 100011 | 011101100 | $= 0xC6EC$ |
| 0 | 011011 | 111001101 | $= 0x37CD$ |
| 0 | 101001 | 000000001 | $= 0x5201$ |

Boolean: Un booleano solo tiene dos valores true y false. Cada booleano ocupa un byte de memoria.

Char (carácter): Un char representa un carácter que ocupa 1 byte de memoria. Los caracteres simples se representan con comillas simples 'a' y para múltiples caracteres o strings se representan con comillas dobles "Hola!".

Recordar que los caracteres se almacenan como números usando la codificación ASCII, lo que significa que es posible hacer operaciones aritméticas con los caracteres.

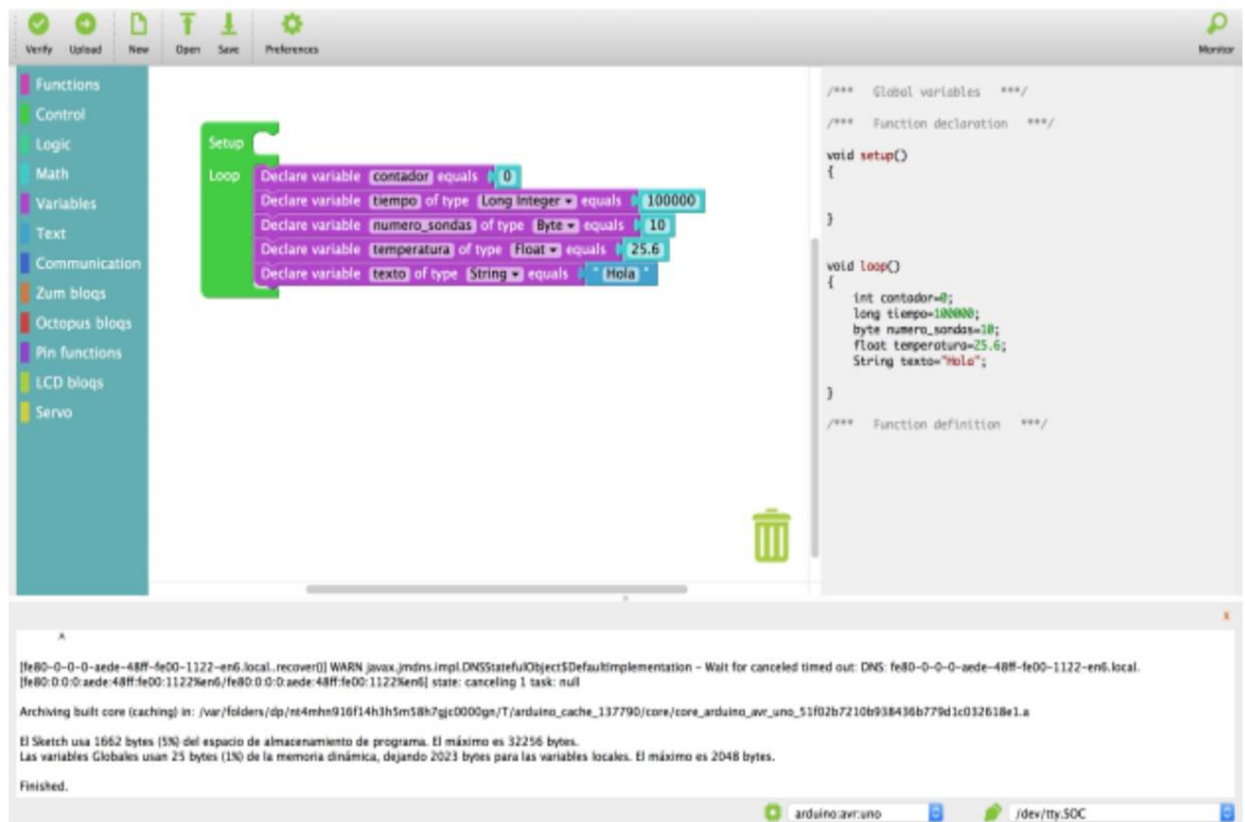
Sistemas de codificación utilizados:

- Binario.
- BCD (Binario codificado a decimal)
- Hexadecimal.
- ASCII

Tipos de Datos en Visualino:

En Visualino podemos definir los tipos de datos tanto en variables locales como globales:

- int
- long
- byte
- float
- String



Conversiones de tipos (Casting)

En ocasiones es necesario forzar el cambio de tipo de dato (casting). Podemos usar las siguientes funciones:

1. char()

2. byte()

3. int()

4. word()

5. long()

6. float()

**2.2.- Declaraciones de variables.**

En C todas las variables han de ser declaradas antes de ser utilizadas. Las variables pueden declararse en tres sitios diferentes:

- Dentro de las funciones (variables locales).
- Fuera de todas las funciones (variables globales).
- En la definición de los parámetros de las funciones.

La declaración tiene el siguiente formato:

<clase><tipo><iden>[=<exp>][,<iden>[=<exp>][...]];

NOTA: lo que ponemos entre corchetes [...] indica opcionalidad.

El <tipo> determina el tipo de dato que almacena la variable.

En <iden> se declara un identificador que opcionalmente puede ser inicializado al valor de

<exp>; se pueden declarar más identificadores separados por comas, pudiendo llevar cada uno su propia inicialización.

El = nos sirve para inicializar las variables dentro de la declaración de tipo. Para hacer esto la declaración debe consistir en un tipo de datos, seguido por un nombre de variable, un signo (=) y una constante del tipo apropiado. Al final se debe poner, como de costumbre, un punto y coma (;).Por tanto, la primera vez que nos aparezca dicha variable, si no ha sido asignada posteriormente a su declaración de tipo, tomará el valor que le hemos dado en dicha declaración.

Ejemplo:

La <clase> determina la forma de almacenamiento de la variable, que determina su visibilidad y su existencia. Existen cuatro formas de almacenamiento:

Auto: Variables locales a una función o a un bloque, es decir, su existencia está ligada a esa función o bloque. La variable se crea en la pila del sistema cuando se invoca la función o cuando se ejecuta el código dentro del bloque, y se destruye cuando acaba la función o bloque. Si no se especifica una clase al declarar una variable, ésta siempre es automática.

Extern: Variables con almacenamiento permanente. Todas las funciones y bloques declarados después de una variable externa podrán acceder a ella. Una variable es de clase externa a una unidad de compilación cuando no se ha definido en esa unidad. En estos casos, el compilador no necesita reservar zonas de memoria para este tipo de variables. El uso de variables externas proporciona un mecanismo adecuado de transferencia de información entre funciones. En particular, podemos transferir información a una función sin usar argumentos. Hemos de distinguir entre definiciones de variables externas y declaraciones de variables externas. Una definición de variable externa se escribe de la misma forma que una variable ordinaria. Tiene que aparecer fuera, y normalmente antes, de las funciones que acceden variables externas. Una declaración de variable externa tiene que empezar por el especificador de tipo de almacenamiento extern. El nombre de la variable externa y su tipo tienen que coincidir con su correspondiente definición de variable externa que aparece en la función. Una declaración de variable externa no puede incluir una asignación de valores iniciales.

Static: Variables que existen desde el comienzo hasta el final de la ejecución del programa.

Una variable de clase estática puede ser global a todo el programa, local a una unidad de compilación, o local a una sola función. Todas las variables globales, por defecto, son de clase estática. Si una variable global se declara explícitamente de clase estática, esta variable se considera como local a una unidad de compilación donde se declara. Una variable local a una función y de clase estática, conserva su valor de una llamada a otra.

Register: Variables que residen en uno de los registros de la CPU. Las variables de tipo

regitro siempre son automáticas y , por tanto, locales a una función. Sólo se puede utilizar la clase registro con los tipos entero y carácter. Se suele usar para contadores.

Ejemplos:

La visibilidad de una variable va a depender del sitio en el que se ha declarado. En función de esto tenemos:

Variables locales: Las variables que se declaran dentro de un bloque de sentencias se denominan variables locales. Estas variables se crean al comienzo del bloque y se destruyen al salir del bloque al que pertenecen.

Variables globales: una variable es global cuando se declara fuera de todos los bloques. Las variables globales se conocen a lo largo de todo el programa y se pueden utilizar desde cualquier sitio. Todas las funciones y bloques declarados después de una variable global, podrán acceder a ella. Si dentro de un bloque se repite la declaración de una variable definida en un bloque exterior, el acceso se referirá exclusivamente a la variable dentro del bloque más interno. Es decir, el nombre de un bloque externo es válido a menos que un bloque interno lo vuelva a definir.

Cuando se declare una variable, como se dijo anteriormente, se le puede asignar un valor inicial (cualquier expresión válida en C), independientemente de que lo mantenga o no a lo largo de todo el programa. Las variables globales y estáticas se inicializan a cero si no se especifica ningún valor. Ambas se deben inicializar con expresiones constantes. Las variables estáticas las inicializa el compilador una sola vez, al comenzar el programa. Las variables

locales y de registro tienen valores desconocidos hasta que se les asigna uno dentro del programa. Si tienen valores iniciales, se asignan cada vez que se ejecuta el bloque donde se definen.

Ejemplos de declaración:

1. int a=4;

2. char asterisco='*';

3. float suma=0.11;

Ejemplos de formas de almacenamiento:

Ejemplo 1:

| int a, b,c; | a, b y c son variables enteras. |
|---|---|
| float raiz1,raiz2; | raiz1 y raiz2 son variables de coma flotante. |
| char indicador; | indicador es una variable de tipo carácter. |
| char texto[80]; | texto es un array de tipo carácter de 80 elementos. Fijémonos en los corchetes que me encierran la especificación del tamaño. |

Ejemplo 2:

int a;

int b;

int c;

float raiz1;

float raiz2;

char indicador;

NOTA: el ejemplo 1 y el ejemplo 2 son equivalentes.

Ejemplo 3:

#include <stdio.h>

void incrementa(); /* función que no devuelve ningún valor void */

/* estoy declarando la función que voy a implementar posteriormente.*/

main()

{

register cont;

for(cont=1;cont <= 3; cont++) { /* Inicializo bucle en la 1ª vuelta a cont=1,*/

printf("Iteración %d\t", cont); /* incrementa en cada vuelta el cont a 1 (cont++)*/ incrementa(); /* repito el cuerpo del bucle mientras cont<=3 */

} /* fin del for */

/* El %d del printf me indica que voy a representar por pantalla un número entero cont, si en lugar de ser cont de tipo entero, fuera de tipo float, y pusiese en el printf un %d, me escribiría por pantalla su parte entera (todo esto de la entrada y salida se verá mejor en el capítulo siguiente).*/

} /* fin del main() (programa principal)*/

void incrementa(){

static int a=1;

int b=1;

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

a++;b++;

} /* fin de la función incrementa */

La salida que dará este programa es:

Iteración 1 a=1 b=1

Iteración 2 a=1 b=1

Iteración 3 a=1 b=1

Ejemplo 4:

Primer archivo:

```
 #include <stdio.h> /*programa simple de varios archivos para escribir "¡Hola!".*/

 extern void salida(void); /*declaración función externa.*/

main()

{

salida();

}
```

Segundo archivo:

```
 extern void salida(void) { /*definición de función externa.*/

printf ("¡Hola!");

return;

 }
```

Observemos como a salida se le asigna el tipo de almacenamiento extern, pues tiene que ser accedida desde otro archivo distinto de aquel en que es definida; por tanto, ha de ser una función externa. Así se incluye la palabra clave extern tanto en la declaración de la función (primer archivo), como en la definición de función (segundo archivo). Como extern es un tipo de almacenamiento por defecto, podíamos haber prescindido de escribir extern tanto en la declaración como en la definición.

## 2.3.- Clases de almacenamiento.

Por defecto, todas las variables llevan asociada una clase de almacenamiento que determinan su accesibilidad y existencia. Estos conceptos de accesibilidad y existencia de las variables pueden alterarse por los calificadores:

Auto.- almacenamiento automático.

Register.- almacenamiento en un registro.

Static.- almacenamiento estático.

Extern.- almacenamiento externo.

Los calificadores auto y register solo se pueden usar con variables locales, el comando

extern sólo se usa con variables globales y el calificador static puede ser utilizado con variables globales y variables locales.

## 2.4.- Sentencias de asignación.

Las sentencias de asignación sirven para calcular expresiones y asignárselas a un dato.

Referencia a un dato = expresión ;

Se debe indicar el dato en el cual se va a guardar el resultado de la expresión, seguido del símbolo = (símbolo de la asignación) y la expresión numérica o lógica a evaluar cuando se ejecute la sentencia. Tras esta sentencia se debe poner siempre el símbolo ;(punto y coma).

En una sentencia de asignación sólo está permitido asignar valores a objetos tales como variables (de cualquier tipo), a una posición de una tabla, o a un elemento de una estructura.

No es posible asignar un valor a una constante, a una función o a un proceso, o, en general, a cualquier expresión numérica o lógica.

Se muestra a continuación un programa con varias asignaciones.

MAIN_PROGRAM_CDIV

BEGIN_PROGRAM

x = x + 1;

angle = (angle*3)/2 - pi/2;

size = (x+y)/2;

z= (abs(x-y)*3) - pow(x, 2);

// ...

END_PROGRAM

Ésta es la forma básica de las asignaciones, si bien existen otros símbolos de asignación que, en lugar de asignar un nuevo valor al dato referido, modifican su valor. Éstos son los símbolos de asignaciones operativas:

+= Suma al dato el resultado de la expresión:

x=2; x+=2; // x = 4

-= Resta al dato el resultado de la expresión

x=4; x-=2; // x = 2

*= Multiplica el dato por el resultado de la expresión

x=2; x*=3; - (x==6)

/= Divide el dato por el resultado de la expresión

x=8; x/=2; // x = 4

%= Pone en el dato el resto de dividir al mismo entre el resultado de la expresión

x=3; x%=2; // x = 1

&= Realiza un AND (binario y/o lógico) entre el dato y el resultado de la expresión y lo asigna como nuevo valor del dato

x=5; x&=6; // x = 4

|= Realiza un OR (binario y/o lógico) entre el dato y el resultado de la expresión y lo asigna como nuevo valor del dato

x=5; x|=6; // x = 7

^= Realiza un OR exclusivo (XOR binario y/o lógico) entre el dato y el resultado de la expresión y lo asigna como nuevo valor del dato

x=5; x^=3; // x = 3

>= Rota a la derecha el dato tantas veces como indique el resultado de la expresión (cada rotación a la derecha equivale a dividir entre 2 el dato)

x=8; x>=2; // x = 2

<= Rota a la izquierda el dato tantas veces como indique el resultado de la expresión (cada rotación a la izquierda equivale a multiplicar por 2 el dato)

x=2; x<=2; // x = 8

También se admiten dentro de la categoría de sentencias de asignación los incrementos y decrementos de un dato. Por ejemplo, si quisiéramos sumarle 1 a la variable local x podríamos hacerlo con la sentencia x=x+1;, con la sentencia x+=1; o bien con el operador de incremento: x++; o ++x;.

Es decir, se aceptan como sentencias de asignación incrementos ( ++ ) o decrementos ( -- ) de un dato.

**2.5.- Definición de constantes**.

Hablamos de constante a un tipo especial de variable (aunque no está bien decir que son variables constantes) que no se puede modificar su valor.

Cuando decimos que no se puede cambiar hablamos que no se puede cambiar durante la ejecución del programa, es decir, en tiempo de ejecución.

Esa es la principal diferencia entre constante y variable. Una variable puede tener cualquier valor (del mismo tipo de datos que hemos declarado), ya sea en tiempo de diseño, lo

cambiamos nosotros en el código fuente, o en tiempo de ejecución, dependiendo de como se está ejecutando el programa.

Pero una constante tendrá su valor inicial, que pondremos en el momento de declararla, siempre.

Ejemplos de constantes en programación

Veamos algunos ejemplos de constantes en programación. Si nos fijamos en las matemáticas, tenemos las dos constantes más famosas que son:

- El número Pi π.
- El número e.

En programación también tenemos otros ejemplos. Éstos pueden venir definidos por el propio lenguaje de programación, como puede ser un código de color, errores del sistema, etc.

Pero también las podemos definir nosotros, como el tamaño máximo de una array o matriz, un texto que se utilice en varios sitios del código…

Como declarar una constante:

Declarar una constante dependerá mucho del lenguaje de programación.

Por ejemplo, para declarar una constante en C se haría así:

```
#define PI 3.1415926
```

En C++, una constante se debe declarar así:

```
const float PI = 3.1416;
```

La mayoría de lenguajes utilizan la palabra reservada const, seguido de su tipo de datos.

En cualquiera de los casos, si intentamos en algún momento modificar su valor:

```
PI = 2;
```

Nos dará un error de compilación y no se ejecutará.

Como ves, las constantes se escriben siempre en mayúscula. Se ha decidido así por estilo de programación, pero independientemente del lenguaje usado, lo habitual es encontrarlo de esta manera para poder identificarlas rápidamente.

A la hora de declararlas, se suele utilizar las mismas condiciones que al declarar una variable, como puede ser:

- No empezar el nombre con un espacio en blanco o un número.
- Algunos lenguajes distinguen mayúsculas y minúsculas, pero otros no. Así que aunque las escribas en mayúsculas, es posible que una variable NO pueda llamarse igual.
- También hay lenguajes de programación que obliga a declarar el tipo de datos en la constante, al igual que las variables.

## 2.6.- Operadores.

Los operadores son símbolos que indican cómo se deben manipular los operandos. Los operadores junto con los operandos forman una expresión, que es una fórmula que define el cálculo de un valor. Los operandos pueden ser constantes, variables o llamadas a funciones, siempre que éstas devuelvan algún valor. El compilador evalúa los operadores, algunos de izquierda a derecha, otros de derecha a izquierda, siguiendo un orden de precedencia. Este orden se puede alterar utilizando paréntesis para forzar al compilador a evaluar primero las partes que se deseen.

**Tipos de operadores:**

1. Aritméticos.
2. Casting.
3. Monarios.
4. Relacionales y Lógicos.
5. Asignación.
6. Condicional.
7. Tratamiento de bits.
8. Punteros.
9. Secuencial.
10. Acceso a estructuras y uniones.
11. [ ] y ( ).

1.- Aritméticos:

| OPERADOR | PROPÓSITO |
| --- | --- |
| + | adición |
| - | sustracción |
| * | multiplicación |
| / | división |
| % | resto de división entera |

NOTA: El % a veces es llamado operador módulo.

El % requiere que sus dos operandos sean enteros, y el segundo no nulo.
El de división (/) requiere que el segundo operando sea no nulo, aunque los operandos no requieren ser enteros. Aquí podemos tener cuatro casos:

1) Si dicha división se produce para dos operandos enteros el resultado será un número entero que será truncado, o sea, cogeremos sólo la parte entera del resultado.

2) Si la división es entre dos números en coma flotante el resultado será otro número en coma flotante.

3) Si la división es entre un número entero y otro en coma flotante, el resultado será un número en coma flotante.
4) Si la división tiene algún operando negativo, o los dos, entonces la división entera (entre dos números enteros) estará truncada hacia cero, esto quiere decir que el resultado será siempre menor en valor absoluto que el verdadero cociente.

Si tenemos 2 operandos, y ambos son tipos de coma flotante con precisión distinta (float y double), el operando de menor precisión (float) se transformará a la precisión del otro (double).
Si tenemos un operando del tipo coma flotante y el otro un char o un int, el char/int, se transformará al tipo de coma flotante.
Si un operando es del tipo long int, y el otro no es del tipo coma flotante (ejemplo int), entonces éste se transformará a long int.

Si ningún operando es del tipo coma flotante, ni long int, ambos se convertirán en int, y el resultado será int.

**2.- Casting:**

El lenguaje permite cambiar el tipo de una variable a través del operador de casting , que se emplea anteponiendo al operando el nombre del tipo requerido, encerrado entre paréntesis: (<tipo>) <operando> donde <tipo> es uno de los tipos básicos, un puntero o un modificador de tipo.

Ejemplo.

Supongamos que i es una variable entera con un valor de 3, y f un variable de coma flotante con un valor de 4.5. La expresión:

(i+f) % 7 sería inválida, ya que el operando (i+f) es de coma flotante en lugar de entero (y el módulo sólo admite enteros), sin embargo utilizando el casting la podemos hacer válida poniendo:

((int)(i+f))% 7

**3.- Monarios:**

Son operadores que actúan sobre un sólo operando para producir un nuevo valor. En C todas las constantes numéricas son positivas. Por tanto, un número negativo es en realidad una expresión, que consiste en el operador monario menos (-), seguido de una constante numérica positiva. No debemos confundir este menos, con el de la operación de la sustracción, el cual requiere dos operandos.

Otros dos operadores monarios interesantes son:

++ Que es el operador incremento, hace que su operando se incremente en uno.

-- Que es el operador decremento, hace que el operando se decremente en uno.

Los operadores incremento y decremento pueden ser utilizados de dos formas diferentes, dependiendo de donde se escriba el operador, antes o después del operando. Si el operador precede al operando (Ej: ++i), el valor del operando se modificará antes de que se utilice con otro propósito. Si el operador sigue al operando (Ej: i--), el valor del operando se modificará después de ser utilizado.

NOTA: El operador incremento y el de decremento son operadores monarios y de asignación, por tanto, en el apartado Asignación los volvemos a mencionar.

Otro operador monario digno de mención es el operador de tamaño: sizeof. Éste devuelve la longitud en bytes del operando, que puede ser una variable o un especificador de tipo (en este caso, debe ir encerrado entre paréntesis).

Ejemplos.

Supongamos que i es una variable entera, y f una variable de coma flotante.
sizeof i devuelve 2
sizeof f devuelve 4

Supongamos un valor inicial para i de 8:

++i la i tomará el valor de 9.
Supongamos ahora que j inicialmente tiene un valor de 7:
--j la j tomará el valor de 6.

Por último, decir que cast (visto anteriormente en el apartado Casting), puede ser considerado un operador monario.

**4.- Lógicos y Relacionales:**

Los operadores lógicos y relacionales tratan con valores verdaderos y falsos. Una expresión con operadores lógicos o relacionales devuelven siempre un valor verdadero o falso. El lenguaje interpreta un valor verdadero cuando es diferente de cero, y falso cuando es igual a cero. Los diferentes operadores de este tipo que existen son:

| OPERADOR | PROPÓSITO |
|---|---|
| > | mayor que |
| >= | mayor o igual que |
| < | menor que |
| <= | menor o igual que |
| == | igual |
| != | distinto |
| && | AND lógico |
| \|\| | OR lógico |
| ! | NOT lógico |

NOTA: Los operadores tienen un orden de precedencia, pudiéndose diferenciar de mayor a menor precedencia, en caso de igualdad de precedencia todos se asocian de izquierda a derecha (excepto los monarios, los condicionales, y los de asignación, que lo hacen de derecha a izquierda), esto quiere decir que si la asociatividad es de izquierda a derecha, y se nos presentan operadores con la misma precedencia, comenzaremos a mirar los operadores más a la izquierda, e iremos avanzando hacia la derecha, dicho esto presentamos los distintos grupos de operadores:

☐ monarios: -, ++, --, !, sizeof(tipo).

☐ Multiplicación (*), división (/) y resto (%) aritméticos.

☐ Suma (+), y sustracción (-) aritméticas.

☐ <, <=, >, >=.

☐ ==, !=.

☐ && (AND lógico).

☐ || (OR lógico).

☐ Operador condicional (se verá posteriormente).

☐ Operador de asignación (se verá posteriormente).

Dado que el lenguaje interpreta como falso el valor cero, y como verdadero cualquier valor diferente de cero, se pueden emplear operadores aritméticos en las expresiones lógicas y de comparación.

Ejemplo:

Supongamos que i es una variable entera con valor de 7, f una variable de coma flotante con valor de 5.5 y c una variable de carácter que representa la 'w'.

<table>
<tr><th>EXPRESIÓN</th><th>INTERPRETACIÓN</th><th>VALOR</th></tr>
<tr><td>f>5</td><td>cierto</td><td>1</td></tr>
<tr><td>(i+f)<=10</td><td>falso</td><td>0</td></tr>
<tr><td>(i>=6)&&(c=='w')</td><td>cierto</td><td>1</td></tr>
<tr><td>(1)c!='p'||i+f<=10</td><td>cierto</td><td>1</td></tr>
<tr><td>!(f>5)</td><td>falso</td><td>0</td></tr>
<tr><td>i>(f+1)</td><td>cierto</td><td>1</td></tr>
<tr><td>(2)i>=6&&c=='w'</td><td>cierto</td><td>1</td></tr>
<tr><td>!(i>(f+1))</td><td>falso</td><td>0</td></tr>
<tr><td>c!='p'</td><td>cierto</td><td>1</td></tr>
<tr><td>(f<11)&&(i>100)</td><td>falso</td><td>0</td></tr>
</table>

(1): Observemos como en este ejemplo se pone de manifiesto la precedencia superior del != (distinto) sobre el OR lógico (||), y del menor o igual (<=) sobre el OR lógico, y a su vez la precedencia del + sobre el <= también queda de manifiesto, por todo ello no nos hace falta poner: (c!='p')||((i+f)<=10).

(2): De la misma forma que en (1), prescindimos de los paréntesis ya que la precedencia nos ahorra dicho trabajo.

**5.- Asignación:**

Se utiliza el operador =, dentro de una asignación, para asignar valores a una variable.

Tiene el siguiente formato: <variable>=<expresión> también admite la forma: <variable><operador>=<expresión> que equivale a: <variable>=<variable><operador><expresión> donde <operador> es cualquiera de los operadores binarios. Por tanto, se tiene los siguientes operadores de asignación:

<table>
<tr><th>OPERADOR</th><th>PROPÓSITO</th></tr>
<tr><td>++</td><td>incremento unario</td></tr>
<tr><td>--</td><td>decremento unario</td></tr>
<tr><td>=</td><td>asignación simple</td></tr>
<tr><td>*=</td><td>asignación de la multiplic.</td></tr>
<tr><td>/=</td><td>asignación de la división</td></tr>
<tr><td>%=</td><td>asignación del resto</td></tr>
<tr><td>+=</td><td>asignación de la suma</td></tr>
<tr><td>-=</td><td>asignación de la resta</td></tr>
<tr><td><<=</td><td>asignación del desplazam. a la izquierda</td></tr>
<tr><td>>>=</td><td>asignación del desplazam. a la derecha</td></tr>
<tr><td>&=</td><td>asignación de la operación AND</td></tr>
<tr><td>|=</td><td>asignación de la operación OR</td></tr>
<tr><td>^=</td><td>asignación de la operación XOR</td></tr>
</table>

Ejemplo:

Supongamos que i y j son variables enteras, con los valores de 5 y 7, respectivamente; f y g variables de coma flotante con valores de 5.5 y -3.25 respectivamente; x,y y z, tienen los valores 2, 3 y 4 respectivamente:

| EXPRESIÓN | EXPRESIÓN EQUIVALENTE | VALOR FINAL |
|---|---|---|
| i+=5 | i=i+5 | 10 |
| f-=g | f=f-g | 8.75 |
| j*=(i-3) | j=j*(i-3) | 14 |
| f/=3 | f=f/3 | 1.833333 |
| i%=(j-2) | i=i%(j-2) | 0 |
| i=3.3 | | 3 (ya que i es variable entera). |
| i=-3.9 | | -3 (ya que i es variable entera) |
| (1)i='x' | | 120 |
| (1)i=('x'-'0')/3 | | 24 |
| (2)x*=-2*(y+z)/3 | x=x*(-2*(y+z)/3) | 8 |

Cuando se evalúa una expresión se obtiene un tipo de resultado que depende de los operandos. Si un operador tiene operandos de tipos diferentes, éstos se convierten a tipo según una jerarquía preestablecida. Las conversiones aritméticas implícitas se realizan mediante la siguiente secuencia, y en ese orden:

- char y short se convierten a int.
- Si un operando es long double, el otro se convierte en long double, y el resultado también.
- En otro caso, si un operando es double, el otro se convierte en double, y el resultado también.

- En otro caso, si un operando es float, el otro se convierte en float, y el resultado también.
- En otro caso, si un operando es unsigned long, el otro se convierte en unsigned
- long, y el resultado también.
- En otro caso, si un operando es long, el otro se convierte en long, y el resultado también.
- En otro caso, si un operando es unsigned, el otro se convierte en unsigned, y el resultado también.
- En otro caso, los operandos son de tipo int, y el resultado es de tipo int.

IMPORTANTE: Cuando se realiza una asignación de variables de diferentes tipos, el valor del lado derecho se ajusta al tipo de variable del lado izquierdo de la expresión.

**6.- Condicional:**

Este operador (?:) se utiliza para reemplazar estructuras sencillas de decisión. Si se desea que una instrucción determinada se ejecute según una cierta condición, se emplea este operador ternario de la siguiente forma: <condición>?<expresión-v>: <expresión-f>

Primero se evalúa la <condición>.

Si es verdadera, se evalúa la <expresión-v>.

Si es falsa, se evalúa la <expresión-f>.

Ejemplo:

- Supongamos que m=50, ¿cuál será el valor asignado a n en la expresión siguiente? n=(m==99?1:2)

En este ejemplo, se nos está diciendo que Si m es igual a 99, tendríamos que coger 1 (que es el que forma la <expresión-v>); si por el contrario m es distinto de 99, entonces tendríamos que coger la <expresión-f> (que en este caso tiene el valor de 2). Como se da el segundo paso, cogeríamos el 2, por tanto, a n se le asigna el valor de 2 (n=2).

- Supongamos que f y g son variables de coma flotante en la siguiente expresión condicional.

  (f<g)?f:g

Esta expresión condicional toma el valor de f, si f es menor que g; de otra forma la expresión condicional tomará el valor de g. En otras palabras, la expresión condicional devuelve el valor de la menor de las dos variables.

- Supongamos que a, b y c son variables de tipo entero, en este ejemplo aparecen seis grupos de precedencia distintos: c+=(a>0 && a<=10)?++a:a/b; (esto es un sentencia que la explicamos debajo de este ejemplo).

La sentencia comienza por la evaluación de la expresión compuesta
(a>0 && a<=10)
Si esta expresión es cierta evaluamos la expresión: ++a, si es falsa evaluamos la expresión a/b. Por último tenemos que hacer la operación de asignación (+=), haciendo que se incremente c en el valor que he obtenido de la expresión condicional.
Si a, b y c poséen los valores 1, 2 y 3, respectivamente, entonces, al evaluar la expresión condicional obtendremos el valor de 2 (ya que evaluaremos la expresión ++a), y posteriormente a c se le asignará el valor de 5 (ya que la asignación sería: c=3+2).

**7.- Tratamiento de Bits:**

El C proporciona un grupo de operadores de manipulación de bits. Estos operadores son:

| OPERADOR | PROPÓSITO |
|---|---|
| & | operación AND. |
| \| | operación OR. |
| ^ | operación XOR. |
| >> | desplazam. a la derecha. |
| << | desplazam. a la izquierda. |

Estos operadores sólo pueden usarse con los tipos int y char y funcionan bit a bit. El operador de desplazamiento se puede utilizar para realizar multiplicaciones o divisiones rápidas, pues cada desplazamiento a la izquierda multiplica por 2, y cada desplazamiento a la derecha divide por 2.

El C distingue entre desplazamientos aritméticos y lógicos:

- Los desplazamientos aritméticos se realizan sobre tipos enteros y mantienen el signo (el bit más alto). o Con los desplazamientos a la izquierda el bit no se altera. o Con los desplazamientos a la derecha el bit se copia en la posición siguiente, con la intención de mantener el signo del dato.
- Los desplazamientos lógicos se realizan sobre datos unsigned. No tienen ninguna consideración particular con el signo.

**8.- Punteros:**

En temas posteriores entraremos más a fondo sobre ellos, ya que son una parte imprescindible del lenguaje. De ellos sólo diremos que los operadores unitarios * y & se utilizan con punteros. Cuando se evalúa el operador unitario &, se obtiene la dirección del operando. El operador *, al evaluarse, devuelve el valor de la variable a la que apunta el operando; por lo tanto, el operando debe ser un puntero.

**9.- Secuencial:**

El operador secuencial (,) se utiliza para concatenar varias expresiones. El lado izquierdo de la coma siempre se evalúa primero.

Ejemplo:

m=(n=30,n++,n*2)

Asigna 30 a la variable n, después incrementa n, para posteriormente multiplicar el resultado del incremento por 2, valor que es asignado a m. El valor de m será 62.

**10-. Acceso a estructuras y uniones:**

Este apartado, al igual que en el de los Punteros, lo trataremos en un tema posterior. Sólo decir que los operadores. Y -> se utilizan para acceder a los campos de uniones o de estructuras. Cuando un puntero apunta a una estructura, se emplea el operador -> para acceder a los campos, y el operador. Se utiliza en otro caso.

**11.- [ ] y ( ).**

Los corchetes se utilizan para acceder a los diferentes elementos de una matriz (se verá posteriormente en temas sucesivos). Para seleccionar un elemento se necesita un índice que

indique la posición a utilizar. Los paréntesis se pueden utilizar dentro de las expresiones para modificar el orden de evaluación predeterminado por el lenguaje.

**2.7.- Procedimientos definidos de entrada y salida.**

SIMPLEIO.C

```
#include "stdio.h" /* cabecera standar para e/s */
main()
{
char c;
printf("Teclee cualquier caracter, X = parar el programa.\n"); do {
c = getchar(); /* toma un caracter simple del teclado */
putchar(c); /* visualiza el caracter en el monitor */
putchar("c");
} while (c != 'X'); /* mientras no se pulse X */
printf("\nFin del programa.\n");
}
```

Llamamos ENTRADA/SALIDA STANDARD(E/S a partir de ahora, para abreviar), a los sitios donde los datos se toman del teclado y se muestran en la pantalla del monitor. Dado que estos dispositivos, teclado y monitor se usan muy a menudo, no necesitan ser mencionados en las instrucciones de E/S. Esto tomará sentido cuando empecemos a usarlos en el programa ejemplo.

Lo primero que advertimos es la primera línea del fichero, «#include stdio.h«. Se parece mucho a la línea #define, ya estudiada anteriormente, excepto en que hay una pequeña modificación y el fichero entero es leído en esta sección. El sistema encontrará el fichero denominado «stdio.h» y leerá el total de su contenido, sustituyendo el mandato correspondiente al #define, por uno adecuado al #include. Obviamente el contenido de «stdio.h» es código válido para C, y por tanto, compilable como parte del programa. Este fichero está compuesto por varias #defines standard, para definir algunas operaciones de E/S standard. Se llama a este fichero desde la cabecera del programa.

Cada cabecera de esta forma tiene un propósito específico, y cualquiera de ellos pueden incluirse en el programa.

La mayoría de compiladores C utilizan el doble signo de » para indicar que un fichero «include» será encontrado en el directorio actual. Un signo «menor que (<)» o un signo «mayor que (>)» indica que el fichero se encontrará en una cabecera de fichero standard. Prácticamente todos los compiladores bajo entorno MS-DOS usan este signo y muchos requieren el fichero «include» presente en el directorio actual.

OPERACIONES DE E/S EN C

Por el momento no tenemos operaciones de E/S definidas como parte del lenguaje, por tanto debemos crearlas.

Dado que nadie ha reinventado sus propias rutinas de E/S las que los programadores crearon son las que se usan habitualmente, un buen número de operaciones de E/S que nos ayudan en nuestros programas. Estas funciones ya se consideran standard, y vienen en la mayoría de compiladores. De hecho, la industria del lenguaje C se ha fijado en la definición dada en el libro de Kernigan y Ritchie.(K&R).

OTROS FICHEROS INCLUDE

Cuando se escriban programas largos y se tenga la necesidad de partirlos en varios ficheros compilados por separado se tendrá ocasión de utilizar mandatos comunes a cada una de las porciones de programa. Sería una ventaja hacer ficheros separados, contiendo el mandato y, usando un #include para insertar cada uno en los ficheros. Si se necesita modificar alguno de estos mandatos, sólo deberá hacerse en un bloque, y automáticamente la modificación se realizará en los restantes. Esto queda lejos, pero es una idea de como usar la directiva «#include».

VOLVAMOS AL FICHERO «SIMPLEIO.C»

Continuemos examinando el fichero. Se define la variable «c» y aparece un mensaje por pantalla, hecho con un printf ya familiar. Nos encontramos en un bucle de tantos pasos mientras no se cumpla la condición, que «c» sea igual a X mayúscula. Las 2 nuevas funciones incluidas en el bucle son de la mayor importancia en este programa, ya que son una novedad. Estas rutinas leen caracteres desde teclado y los muestran por el monitor, uno a uno.

La función «getchar()» lee un caracter del dispositivo de entrada standard (normalmente el teclado), y lo asigna a la variable «c». La siguiente función, «putchar()» usa el dispositivo standard de salida (el monitor, en este caso), para mostrar el contenido de la variable «c».
El carácter es visualizado en la posición actual del cursor y este avanza un espacio. El sistema se encarga de hacer muchas cosas por nosotros. El bucle continúa leyendo y mostrando caracteres mientras no tecleemos X para finalizar.
Compilando y ejecutando el programa se verá que todo lo que se teclee aparecerá por pantalla inmediatamente después de pulsar ENTER.

D.O.S NOS AYUDA (O NOS INDICA EL CAMINO)

Debemos saber un poco acerca del DOS y sus procesos para entender que pasa aquí.
Cuando el dato es leído desde el teclado, bajo el control del sistema operativo, los caracteres se almacenan en un buffer (reserva de memoria con un fin específico), mientra no se pulse ENTER, ya que entonces el total de caracteres almacenados pasará al programa. En el ejemplo, a medida que se introducen los caracteres, van apareciendo por pantalla. A esto se le llama ECO, y sucede en algunas aplicaciones que utilizamos habitualmente.

Con el anterior párrafo en mente, debería quedar claro que cuando se está tecleando una línea de datos en «SIMPLEIO.C» los caracteres están siendo reflejados por el DOS y que cuando se pulsa ENTER, el conjunto de estos caracteres pasa al programa. A medida que se introduce un caracter, éste aparece por pantalla, con lo cual en la pantalla se forma una repetición de la línea tecleada. Para entender mejor esto, tecleemos una línea, con una X en mayúsculas, en alguna parte de la línea, en medio, por ejemplo. Podemos escribir tantos caracteres después de X como deseemos, y aparecerán en el monitor debido a que el
DOS los repite, y los mete en el buffer. DOS no sabe que X es el caracter finalizador.

Cuando la línea de texto se pasa al programa, los caracteres son aceptados uno a uno, hasta llegar a X. Tras esto, finaliza el bucle, y el programa. Los caracteres que siguen a X no aparecerán en la línea.

OTRO EXTRAÑO MÉTODO DE E/S

SINGLEIO.C

```
#include "stdio.h"
main()
{
char c;
printf("Pulse cualquier caracter, acaba el programa con X\n"); do {
c = getch(); /* toma un caracter */
38
putchar(c); /* lo visualiza por pantalla */
} while (c != 'X');
printf("\nFin del programa.\n");
}
```

Otra vez empezamos con una cabecera de E/S, y definimos una variable, «c» e imprimimos un mensaje de bienvenida. Como en el programa anterior, estamos en un bucle que continúa ejecutándose mientras no tecleemos una X mayúscula, pero aquí la acción es sensiblemente distinta.

«getch()» es una nueva función de «captación de caracteres». Es diferente a «getchar()» en el sentido de que no tiene relación con el DOS. Lee los caracteres sin eco, y los manda directamente al programa, donde son procesados al momento. Esta función, entonces, lee un carácter, lo manda al programa, aparece por pantalla y continúa el proceso mientras no aparezca la famosa X.

Cuando el programa se ejecute, las líneas en pantalla no se repetirán cuando se pulse ENTER, y cuando se teclee X, el programa finalizará. No es necesario el retorno de carro para aceptar una línea con X. Pero hay otro problema, no hay línea de retorno«(otro tipo de ENTER) con el ENTER.

AHORA NECESITAMOS UN «LINE-FEED»

Esto no es aparente en muchos programas, pero cuando pulsamos ENTER, el programa añade otra cosa al retorno de carro. Debe volver a la primera posición izquierda de la siguiente línea. Esto, conocido como «alimentación de línea», no es automático.

Necesitamos implementarlo en el programa. En el fichero «BETTERIN.C» encontraremos un cambio para incorporar este elemento.

```
#include "stdio.h"
#define CR 13 /* definición de CR como 13 */
#define LF 10 /* definición de LF como 10 */
main()
{
char c;
printf("Entre algunos caracteres, pulse X para terminar.\n"); do {
c = getch(); /* toma un caracter */
putchar(c); /* lo saca por pantalla */
if (c == CR) putchar(LF); /* si es un retorno de carro inserta una nueva línea */
} while (c != 'X');
printf("\nFin de programa.\n");
}
```

Aquí tenemos dos mandatos adicionales al principio que definen códigos para el «linefeed» y para el retorno de carro. Si miramos cualquier tabla del código ASCII, encontraremos los códigos 10 y 13 definidos exactamente como aquí. En el programa principal, tras mostrar el caracter lo comparamos con retorno de carro, y si es igual, también mostramos un cambio de línea, el line-feed. Podríamos considerar correcto lo que aparece en las #define, «if (c==13) putchar(10);», pero no sería muy descriptiva sobre que hace aquí. El método usado en el programa representa la mejor práctica de programación.

## ¿QUÉ MÉTODO ES MEJOR?

Hemos examinado 2 métodos de leer caracteres en un programa en C, y nos encontramos en la necesidad de elegir uno. Esta elección depende de la aplicación a usar, porque cada método tiene ventajas e inconvenientes. Demos una mirada a ambos.

Cuando usamos el primer método, DOS hace todo el trabajo por nosotros, almacenando los caracteres en un buffer de entrada y señalando cuando una línea ha sido completada.

Podríamos escribir un programa que, por ejemplo, hiciera muchos cálculos y, pidiera datos. Mientras estuviéramos haciendo esos cálculos, DOS acumularía una línea de caracteres para nosotros, y estarían ahí cuando estuviéramos preparados para recibirlos.

No obstante, no podríamos conocer los caracteres del buffer, hasta que no pulsemos ENTER, ya que hasta entonces el DOS considera el buffer incompleto e inaccesible.

El segundo método, usado en BETTERIN.C, nos permite recoger un caracter y mostrarlo inmediatamente. No tenemos que esperar a que DOS llene su buffer. La máquina no hace nada más que esperar a que pulsemos un caracter. Método muy útil para aplicaciones interactivas.

Le corresponde al programador decidir cual es mejor.

Debemos mencionar ahora que existe la función «ungetchar()«, que trabaja como «getch».

Si recoge un carácter, y decide que ha ido demasiado lejos, puede deshacerse de ese carácter y eliminarlo del dispositivo de entrada. Esto simplifica algunos programas porque no sabe que no necesita el carácter mientras no lo recoja. Nosotros sólo podemos «olvidar» un carácter en el dispositivo de entrada, pero esto es suficiente para cumplir el objetivo para el cual la función fue creada. Es complicado demostrar el uso de esta función en un programa, por tanto la estudiaremos cuando nos haga falta.

Trabajando Con Enteros

Ejemplo de lectura en algunos tipos formateados.

INTIN.C.

```
#include "stdio.h" main()
{ int valin; printf("Ingrese un número de 0 a 32767, para con 100.\n"); do { scanf("%d",&valin);
/* Lee un entero y lo asigna a valin */ printf("El valor es %d\n",valin);
} while (valin != 100); printf("Fin de programa\n");
}
```

La estructura de este programa es muy parecida a los últimos 3 ejemplos, excepto en que aquí definimos un entero y, un bucle que funcionará hasta que la variable tome el valor 100.

Tenemos ejemplos de lectura de caracteres simples, en los últimos 3 programas, pero en este leemos un valor entero de una sola vez, usando la función «scanf«. Esta función es muy similar a «printf», que hemos estado usando hasta ahora, excepto en que en esta ocasión se

usa para introducir ejemplos de salida. Examinando la línea con «scanf» vemos que no pide para la variable «valin» directamente, pero da su dirección, dado que espera que la función devuelva un valor. La función necesita tener la dirección de la variable para poder devolver un valor al programa. Ya que nos falta un puntero en «scanf», es probable que encontremos problemas usando esta función.

La función «scanf» examina la línea de entrada, mientras encuentra el primer dato. Ignora los espacios en blanco, y, en este caso, lee datos de tipo entero, mientras no encuentre un espacio en blanco o un carácter decimal no válido, en cuyo caso cesará la lectura y devolverá el valor.

Recordando lo que hablamos acerca del buffer que crea DOS, y como trabaja, deberá quedar claro que no sucede nada hasta que no se encuentre el ENTER. Entonces, el buffer se cierra y, nuestro programa buscará en la línea de datos todos los enteros hasta examinarla totalmente. Esto es así porque estamos en un bucle y le hemos dicho que busque un valor, lo imprima, busque otro, lo imprima, etc. Si entra valores en una línea, leerá cada uno sucesivamente, y los sacará por pantalla. Entrando el valor 100, acabaremos el programa, y entrando el valor 100 junto con otros valores, se causará la finalización del programa antes de la lectura de los valores posteriores a 100.

DA RESPUESTAS A VECES ERRÓNEAS

Si se introduce un número como 32767, o menor, aparecerá correctamente en pantalla, pero si se ingresa un número más alto, aparecerá produciendo un error. Por ejemplo, si entramos el valor 32768, aparecerá como -32768, y tecleando 65535 aparecerá 0. En sí, no son errores graves, pero están causados por la manera en que está definido un entero.

El bit más significativo del patrón de 16 bits, válido para variables enteras es el de signo, por lo tanto sólo hay 15 bits a la izquierda del valor. Este, por tanto, sólo puede tener rango de -32768 a 32767. Cualquier otro valor no comprendido entre estos dos, queda fuera de margen. Debemos tener cuidado con esto en nuestros programas. Es otro ejemplo de la gran responsabilidad que supone programar en C, a cambio de la gran flexibilidad que presta el lenguaje, mayor que en otros de alto nivel, como PASCAL, MODULA-2, etc.

El párrafo anterior es aplicable a la mayoría de compiladores bajo entorno MS-DOS.

Existe una pequeña posibilidad de que el compilador utilizado tenga un rango superior a 16 bits, en cuyo caso los límites deben ser modificados.

Puede ejecutarse el programa probando con diferentes valores en una línea, para ver resultados y con varios números entre blancos; o con números demasiado grandes, a ver qué pasa y, finalmente con algún carácter no válido para ver que hace el sistema con caracteres no decimales.

ENTRANDO CADENAS DE CARACTERES

El siguiente programa es un ejemplo de lectura de una variable de cadena. Este programaes idéntico al anterior, excepto en que en vez de definir una variable entera, hemos definido una de cadena con un límite de 24 caracteres(recordemos que una cadena de caracteres debe tener un carácter EOL). La variable en el «scanf» no necesita un «&» porque «big es una variable array y, por definición, un puntero. Este programa no debe requerir explicaciones adicionales.

STRINGIN.C

```
#include "stdio.h" main()
{ char big[25];
Printf ("Entre una cadena de caracteres, máximo de 25.\n");
Printf ("Una X en la columna 1 finalizara el programa.\n");
Do {
Scanf ("%s",big);
Printf ("La cadena es -> %s\n",big);
} while (big[0] != 'X');
 
Printf ("Fin de programa.\n");
}
```

Ejecutando el programa vemos que parte a las frases en palabras separadas. Cuando usamos la cadena en modo de entrada, «scanf» lee los caracteres de la cadena hasta que llega al final o se topa con un espacio en blanco. Por tanto, si lee una palabra y un espacio en blanco, imprime el resultado. Dado que estamos en un bucle, el programa continúa leyendo palabras hasta que sature el buffer de entrada del DOS. Hemos escrito el programa para que pare

cuando encuentre una X mayúscula en la columna 1, pero ya que la frase se almacena en palabras separadas, el programa se parará cuando encuentre una palabra que empiece en X. Si probamos entrando 5 palabras en una frase, con una X mayúscula en el primer carácter de la tercera palabra, debemos ver las 3 primeras palabras en pantalla y la última, simplemente la ignorará el programa cuando pare.

Entrando más de 24 caracteres puede generarse un error, pero dependerá mucho del sistema que se esté usando. En el programa presente, es responsabilidad del operador contar los caracteres y parar cuando el buffer esté lleno.

Una última observación con las funciones de E/S. Es perfectamente legal mezclar «scanf» y «getchar» durante las operaciones de entrada. De la misma forma, tampoco es incorrecto emplear «printf» y «putchar» juntos.

E/S EN MEMORIA

A continuación vemos otro tipo de E/S que permanece en la memoria del ordenador, la cual no hay que cargar del disco cada vez que la usemos.

INMEM.C

```
Main()
{
Int numbers [5], result[5], index;
Char line [80];
Numbers [0] = 74;
Numbers [1] = 18;
Numbers [2] = 33;
Numbers [3] = 30;
Numbers [4] = 97;
Sprintf (line,"%d %d %d %d %d\n", numbers [0], numbers[1],
Numbers [2], numbers [3], numbers[4]);
Printf ("%s", line);
Sscanf (line,"%d %d %d %d %d",&result[4],&result[3],
(result+2),(result+1),result);
For (index = 0;index < 5;index++)
```

```
Printf ("El resultado final es %d\n",result[index]);
}
```

En «INMEM.C» definimos algunas variables, luego asignamos algunos valores a título de ejemplo a la variable "numbers» y entonces usamos la función «sprintf«. Esta función actúa como un «printf» normal y corriente, excepto en que al contrario que ésta, imprime la línea de manera formateada y la asigna a una variable de cadena, no a la pantalla. En este caso, la cadena va a la variable de cadena «line», ya que esta es la variable que hemos insertado en el primer argumento del «sprintf». Los espacios después del segundo %d fueron puestos ahí para ilustrar la siguiente función, que encontramos en la línea.

Mostramos el resultado, y encontramos la salida idéntica a la que hubiera producido un «sprintf».

Dado que la cadena generada está todavía en memoria, podemos leerla ahora con la función «sscanf«, que lee los datos desde una cadena en memoria, no desde teclado. Le decimos a la función en el primer argumento que «line» es la cadena para entradas, y el resto de la línea es exactamente igual que si hubiéramos empleado «scanf». Es esencial el uso de puntero en los datos, ya que necesitamos recibir los datos de la función. Para ilustrar que existen diferentes maneras de declarar un puntero, se usan varios métodos, pero todos definen lo mismo. Los dos primeros, simplemente declaran la dirección de los elementos del array, mientras que los tres últimos demuestran que «result» sin subíndice es un puntero. Para darle más interés son leídos al revés. Finalmente, los valores aparecen por pantalla.

¿ES REALMENTE ÚTIL?

Parece un poco estúpido leer datos del ordenador, pero tiene un propósito. Es posible leer datos usando cualquiera de las funciones standard, y hacer una conversión de formato en memoria. Podríamos leer en una línea de datos, ver algunos caracteres significativos, usar rutinas de formateo de entradas, para reducir la línea a una representación interna.

ERRORES STANDARD DE SALIDA

A veces es necesario redireccionar la salida del dispositivo standard a un fichero. Pueden necesitarse algunos mensajes para ir al dispositivo, en este caso el monitor. La siguiente función lo permite.

SPECIAL.C

```
#include "stdio.h"
main()
{
int index;
for (index = 0;index < 6;index++) {
printf("Ejemplo de función de salida standard.\n");
fprintf(stderr,” línea de dispositivo de error.\n");
}
exit(4);
}
```

El programa consiste en un bucle con 2 mensajes de salida, uno en el dispositivo standard de salida y el otro a un dispositivo standard de error. El mensaje al dispositivo standard de error aparece con la función «fprintf» e incluye el dispositivo denominado «stderr», como primer argumento. Con algún que otro cambio funciona igual que «printf». Ignoremos la línea con «exit» por el momento, la veremos más adelante.

Ejecutando el programa se verán 12 líneas en la pantalla. Para ver la diferencia, puede ejecutarse otra vez el programa con la salida redireccionada en un fichero, por ejemplo «stuff», entrando desde DOS la siguiente línea: A>special>stuff. Esta vez sólo se verán las
6 líneas del dispositivo de error, y si se examina el directorio, se verán las otras 6 contenidas en el fichero STUFF. Puede usarse el redireccionamiento de E/S en cualquier programa que lo precise. También puede usarse esta técnica para leer datos desde un fichero, como lo veremos más adelante.

## FUNCIONES

### 3.1.- Definición de función.

En computación, una subrutina o subprograma (también llamada procedimiento, función o rutina), como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.

Algunos lenguajes de programación, como Visual Basic.NET o Fortran, utilizan el nombre función para referirse a subrutinas que devuelven un valor.

Desde un punto de vista práctico, podemos decir que una función es una parte de un programa (subrutina) con un nombre, que puede ser invocada (llamada a ejecución) desde otras partes tantas veces como se desee. Un bloque de código que puede ser ejecutado como una unidad funcional. Opcionalmente puede recibir valores; se ejecuta y puede devolver un valor. Desde el punto de vista de la organización, podemos decir que una función es algo que permite un cierto orden en una maraña de algoritmos.

Las funciones son la parte central de la programación. Algunos lenguajes, como Pascal, distinguen entre procedimientos ("Procedures") y funciones. En C++ las funciones desempeñan ambos papeles, aunque en cierto modo, los ficheros desempeñan algunas funcionalidades de lo que, en otros lenguajes como Modula-2, se denominan módulos.

Otra diferencia substancial es que C++ no permite el anidamiento de funciones, es decir, definir funciones dentro de otras. En C++ todas las funciones se definen a nivel de fichero, con lo que tienen ámbito global al fichero.

Existe una excepción a esta regla; se refiere a las funciones miembro de las clases, que pueden ser declaradas y definidas dentro de las propias clases. Aunque las clases no son por supuesto funciones en el sentido estricto, si representan cierta compartimentación de datos y procedimientos (un tipo de "módulos").

### 3.2.- Llamada de una función.

Se puede acceder (llamar) a una determinada función desde cualquier parte de un programa. Cuando se llama a una función, se ejecutan las instrucciones que constituyen dicha función. Una vez que se ejecutan las instrucciones de la función, se devuelve el control del programa a la siguiente instrucción (si existe) inmediatamente después de laque provocó la llamada a la función.

Cuando se accede a una función desde un determinado punto del programa, se le puede pasar información mediante unos identificadores especiales conocidos como argumentos (también denominados parámetros). Una vez que la función procesa esta información, devuelve un valor mediante la instrucción return. La estructura general de una función en

C es la siguiente:

tipo_de_retorno nombre_de_la_función (lista_de_parámetros) {

cuerpo_de_la_función

return expresión

}

Donde:

• tipo_de_retorno: es el tipo del valor devuelto por la función, o, en caso de que la función no devuelva valor alguno, la palabra reservada void.

• nombre_de_la_función: es el nombre o identificador asignado a la función.

• lista_de_parámetros: es la lista de declaración de los parámetros que son pasados a la función. Éstos se separan por comas. Debemos tener en cuenta que pueden existir funciones que no utilicen parámetros.

• cuerpo_de_la_función: está compuesto por un conjunto de sentencias que llevan a cabo la tarea específica para la cual ha sido creada la función.

• return expresión: mediante la palabra reservada return, se devuelve el valor de la función, en este caso representado por expresión.

Vamos a suponer que queremos crear un programa para calcular el precio de un producto basándose en el precio base del mismo y el impuesto aplicable. A continuación mostramos el código fuente de dicho programa:

```
#include <stdio.h>
float precio(float base, float impuesto); /* declaración */
main()
{
float importe = 2.5;
float tasa = 0.07;
printf("El precio a pagar es: %.2f\n", precio(importe, tasa));
return 0;
}
float precio(float base, float impuesto) /* definición */
{
```

float calculo; calculo = base + (base * impuesto); return calculo;

}

El ejemplo anterior se compone de dos funciones, la función requerida main y la función creada por el usuario precio, que calcula el precio de un producto tomando como parámetros su precio base y el impuesto aplicable. La función precio calcula el precio de un producto sumándole el impuesto correspondiente al precio base y devuelve el valor calculado mediante la sentencia return.

Por otra parte, en la función main declaramos dos variables de tipo float que contienen el precio base del producto y el impuesto aplicable. La siguiente sentencia dentro de la función main es la llamada a la función de biblioteca printf, que recibe como parámetro una llamada a la función precio, que devuelve un valor de tipo float. De esta manera, la función printf imprime por la salida estándar el valor devuelto por la función precio. Es importante tener en cuenta que las variables importe y tasa (argumentos) dentro de lafunción main tienen una correspondencia con las variables base e impuesto (parámetros) dentro de la función precio respectivamente.

### 3.3.- Declaración de una función ( FORWAR).

Antes de empezar a utilizar una función debemos declararla. La declaración de una función se conoce también como prototipo de la función. En el prototipo de una función se tienen que especificar los parámetros de la función, así como el tipo de dato que devuelve. Los prototipos de las funciones que se utilizan en un programa se incluyen generalmente en la cabecera del programa y presentan la siguiente sintaxis:

tipo_de_retorno nombre_de_la_función(lista_de_parámetros);

En el prototipo de una función no se especifican las sentencias que forman parte de la misma, sino sus características. Por ejemplo: int cubo(int numero);

En este caso se declara la función cubo que recibe como parámetro una variable de tipo entero (numero) y devuelve un valor del mismo tipo. En ningún momento estamos especificando qué se va a hacer con la variable numero, sólo declaramos las características de la función cubo.

Cabe señalar que el nombre de los parámetros es opcional y se utiliza para mejorar la comprensión del código fuente. De esta manera, el prototipo de la función cubo podría expresarse de la siguiente manera: int cubo(int);

Los prototipos de las funciones son utilizados por el compilador para verificar que se accede a la función de la manera adecuada con respecto al número y tipo de parámetros, y al tipo de valor de retorno de la misma. Veamos algunos ejemplos de prototipos de funciones: int potencia (int base, int exponente);

double area_rectangulo (float base, float altura); int mayor(int, int);

struct direccion entrada(void);

Las funciones de biblioteca se declaran en lo que se conocen como ficheros de cabecera o ficheros .h (del inglés headers, cabeceras). Cuando deseamos utilizar alguna de las funciones de biblioteca, debemos especificar el fichero .h en que se encuentra declarada la función, al inicio de nuestro programa. Por ejemplo, si deseamos utilizar la función printf en nuestro programa, debemos incluir el fichero stdio.h que contiene el prototipo de esta función.

**3.4.- Pasos de parámetros de una función.**

Tras declarar una función, el siguiente paso es implementarla. Generalmente, este paso se conoce como definición. Es precisamente en la definición de una función donde se especifican las instrucciones que forman parte de la misma y que se utilizan para llevar a cabo la tarea específica de la función. La definición de una función consta de dos partes, el encabezado y el cuerpo de la función. En el encabezado de la función, al igual que en el prototipo de la misma, se tienen que especificar los parámetros de la función, si los utiliza y el tipo de datos que devuelve, mientras que el cuerpo se compone de las instrucciones necesarias para realizar la tarea para la cual se crea la función. La sintaxis de la definición de una función es la siguiente:

tipo_de_retorno nombre_de_la_función (lista_de_parámetros)

{ sentencias; }

El tipo_de_retorno representa el tipo de dato del valor que devuelve la función. Este tipo debe ser uno de los tipos simples de C, un puntero a un tipo de C o bien un tipo struct.

De forma predeterminada, se considera que toda función devuelve un tipo entero (int). En otras palabras, si en la declaración o en la definición de una función no se especifica el

tipo_de_retorno, el compilador asume que devuelve un valor de tipo int. El nombre_de_la_función representa el nombre que se le asigna a la función. Se recomienda que el nombre de la función esté relacionado con la tarea que lleva a cabo. En caso de que la función utilice parámetros, éstos deben estar listados entre paréntesis a continuación del nombre de la función, especificando el tipo de dato y el nombre de cada parámetro. En caso de que una función no utilice parámetros, se pueden dejar los paréntesis vacíos o incluir la palabra void, que indica que la función no utiliza parámetros. Después del encabezado de la función, debe aparecer, delimitado por llaves ({ y }), el cuerpo de la función compuesto por las sentencias que llevan a cabo la tarea específica de la función.

Veamos la definición de la función cubo declarada en el apartado anterior: int cubo(int base) { int potencia; potencia = base * base * base; return potencia; }

Como ya hemos visto, a los argumentos que recibe la función también se les suele llamar parámetros. Sin embargo, algunos autores consideran como parámetros a la lista de variables entre paréntesis utilizada en la declaración o en la definición de la función, y como argumentos los valores utilizados cuando se llama a la función. También se utilizan los términos argumentos formales y argumentos reales, respectivamente, para hacer esta distinción. Cuando un programa utiliza un número elevado de funciones, se suelen separar las declaraciones de función de las definiciones de las mismas. Al igual que con las funciones de biblioteca, las declaraciones pasan a formar parte de un fichero cabecera (extensión .h), mientras que las definiciones se almacenan en un fichero con el mismo nombre que el fichero .h, pero con la extensión .c. En algunas ocasiones, un programador no desea divulgar el código fuente de sus funciones. En estos casos, se suele proporcionar al usuario el fichero de cabecera, el fichero compilado de las definiciones (con extensión .o, de objeto) y una documentación de las mismas. De esta manera, cuando el usuario desea hacer uso de cualquiera de las funciones, sabe qué argumentos pasarle y qué tipo de datos devuelve, pero no tiene acceso a la definición de las funciones.

3.5.- Funciones predefinidas en “C”.

En la bilioteca estándar de C++ existen muchas funciones ya creadas, que nos pueden resultar útiles.

Dentro del fichero de cabecera "math.h" tenemos acceso a muchas funciones matemáticas predefinidas en C, como:

- acos(x): Arco coseno
- asin(x): Arco seno
- atan(x): Arco tangente
- atan2(y,x): Arco tangente de y/x (por si x o y son 0)
- ceil(x): El valor entero superior a x y más cercano a él
- cos(x): Coseno
- cosh(x): Coseno hiperbólico
- exp(x): Exponencial de x (e elevado a x)
- fabs(x): Valor absoluto
- floor(x): El mayor valor entero que es menor que x
- fmod(x,y): Resto de la división x/y
- log(x): Logaritmo natural (o neperiano, en base "e")
- log10(x): Logaritmo en base 10
- pow(x,y): x elevado a y
- sin(x): Seno
- sinh(x): Seno hiperbólico
- sqrt(x): Raíz cuadrada
- tan(x): Tangente
- tanh(x): Tangente hiperbólica (todos ellos usan parámetros X e Y de tipo "double") y una serie de constantes como
- M_E, el número "e", con un valor de 2.71828...
- M_PI, el número "Pi", 3.14159...

La mayoría de ellas son específicas para ciertos problemas matemáticos, especialmente si interviene la trigonometría o si hay que usar logaritmos o exponenciales. Pero vamos a destacar las que sí pueden resultar útiles en situaciones más variadas:

- La raiz cuadrada de 4 se calcularía haciendo x = sqrt(4);
- La potencia: para elevar 2 al cubo haríamos y = pow(2, 3);

- El valor absoluto: si queremos trabajar sólo con números positivos usaríamos n = fabs(x);

Así se podría calcular el coseno de un ángulo:

```
#include <iostream>
#include <cmath>
using namespace std; int main()
{
 float anguloGrados = 45;
 float PI = 3.14159265;
 float anguloRadianes = anguloGrados * PI / 180;
 cout << "El coseño de 45 grados es "
 << cos(anguloRadianes) << endl;  return 0; }
```

En un programa de gestión o una utilidad que nos ayuda a administrar un sistema, no es habitual que podamos permitir que las cosas ocurran al azar. Pero los juegos se encuentran muchas veces entre los ejercicios de programación más completos, y para un juego sí suele ser conveniente que haya algo de azar, para que una partida no sea exactamente igual a la anterior.

Generar números al azar ("números aleatorios") usando C no es difícil. Si nos ceñimos al estándar ANSI C, tenemos una función llamada "rand()", que nos devuelve un número entero entre 0 y el valor más alto que pueda tener un número entero en nuestro sistema.

Generalmente, nos interesarán números mucho más pequeños (por ejemplo, del 1 al 100), por lo que "recortaremos" usando la operación módulo ("%", el resto de la división).

Vamos a verlo con algún ejemplo:

☐ Para obtener un número del 0 al 9 haríamos x = rand() % 10;

☐ Para obtener un número del 0 al 29 haríamos x = rand() % 30;

☐ Para obtener un número del 10 al 29 haríamos x = rand() % 20 + 10;

☐ Para obtener un número del 1 al 100 haríamos x = rand() % 100 + 1;

☐ Para obtener un número del 50 al 60 haríamos x = rand() % 11 + 50;

☐ Para obtener un número del 101 al 199 haríamos x = rand() % 100 + 101;

Pero todavía nos queda un detalle para que los números aleatorios que obtengamos sean

"razonables": los números que genera un ordenador no son realmente al azar, sino "pseudo-aleatorios", cada uno calculado a partir del siguiente. Podemos elegir cual queremos que sea el primer número de esa serie (la "semilla"), pero si usamos uno prefijado, los números que se generarán serán siempre los mismos. Por eso, será conveniente que el primer número se base en el reloj interno del ordenador: como es casi imposible que el programa se ponga en marcha dos días exactamente a la misma hora (incluyendo milésimas de segundo), la serie de números al azar que obtengamos será distinta cada vez.

La "semilla" la indicamos con "srand", y si queremos basarnos en el reloj interno del ordenador, lo que haremos será srand(time(0)); antes de hacer ninguna llamada a "rand()".

Para usar "rand()" y "srand()", deberíamos añadir otro fichero a nuestra lista de "includes", el llamado "stdlib":

```
#include <cstdlib>
```

Si además queremos que la semilla se tome a partir del reloj interno del ordenador (que es lo más razonable), deberemos incluir también "time":

```
#include <ctime>
```

Vamos a ver un ejemplo, que muestre en pantalla un número al azar entre 1 y 10:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std; int main()
{  int n;
 srand(time(0));
 n = rand() % 10 + 1;
 cout << "Un número entre 1 y 10: " << n << endl;  return 0; }
```

**3.6.- Recursividad.**

Recursión (ciencias de computación) La mayoría de los lenguajes de programación dan soporte a la recursión permitiendo a una función llamarse a sí misma desde el texto del programa. Los lenguajes imperativos definen las estructuras de loops como while y for que son usadas para realizar tareas repetitivas.

La recursividad es un método que lo podemos usar ya que se llama a sí mismo. En otras palabras, la recursividad nos ayuda a que los procesos se repitan así mismos y sea más fácil su codificación. La recursividad la podemos usar cuando tenemos problemas matemáticos como la derivada de cualquier función

Consiste en:

- En el cuerpo de sentencias del subalgoritmo se invoca al propio subalgoritmo para resolver "una versión más pequeña" del problema original.
- Habrá un caso (o varios) tan simple que pueda resolverse directamente sin necesidad de hacer otra llamada recursiva.
- Aspecto de un subalgoritmo recursivo. ALGORITMO Recursivo(...) INICIO ... Recursivo(...); ...

**3.7.- Estructuras de control.**

DE SELECCIÓN. Las estructuras de control de selección, ejecutan un bloque de instrucciones u otro, o saltan a un subprograma o subrutina según se cumpla o no una condición.

- Estructura de control. Las estructuras de control permiten tomar decisiones y realizar un proceso repetidas veces. Son las encargadas de controlar el flujo de un programa.
- Selección if simple. Se trata de una estructura de control que permite desviar un curso de acción según la condición simple, sea falsa o verdadera. Si la condición es verdadera, se ejecuta el la condición 1, de lo contrario, se ejecuta la condición 2.Se pueden plantear múltiples condiciones simultáneamente; ejemplo, si se cumple la (Condición 1), se ejecuta (sentencias 1) en caso contrario se comprueba la (Condición 2), si es cierta se ejecuta (sentencias 2), si ninguna de ellas es cumple se ejecuta (sentencias else).
- Select-Case. Esta sentencia permite ejecutar una de entre varias acciones en función del valor de una expresión. Es una alternativa a if then else cuando se compara la misma expresión con diferentes valores.

1. Se mira la expresión, dando como resultado un número.

2. Luego, se recorren los "Case" dentro de la estructura buscando que el número coincida con uno de los valores.

3. Es necesario que coincidan todos sus valores.

4. Cuando se encuentra, se ejecuta el bloque de sentencias correspondiente y se sale de la estructura Select-Case.

5. Si no se encuentra ninguna coincidencia se ejecuta el bloque de sentencias de la sección "Case Else".

DE CONTROL ITERATIVAS. Las estructuras de control iterativas inician o repiten un bloque de instrucciones si se cumple una condición o mientras se cumple una condición.

- Do-While. Mientras la condición sea verdadera, se ejecutarán las sentencias.
- Do-Until. Se ejecuta el bloque de sentencias, hasta que la condición sea verdadera
- For-Next. La sentencia For da lugar a un lazo o bucle, y permite ejecutar un conjunto de sentencias cierto número de veces.

1. Primero, se evalúan las expresiones 1 y 2, dando como resultado dos números.

2. La variable del bucle recorrerá los valores desde el número dado por la expresión
1 hasta el número dado por la expresión 2.

3. Las sentencias se ejecutará en cada uno de los valores que tome la variable del bucle.

ESTRUCTURAS ANIDADAS. Las estructuras de control básicas pueden anidarse, ponerse una dentro de otra.

- Estructura For-Next dentro de una estructura If-Then-Else.

IF A > B THEN

FOR X = 1 To 5

(Bloque de sentencias 1)

NEXT

ELSE

(Bloque de instrucciones 2)

END IF

SI A> B ENTONCES

Para x = 1 a 5

(Bloque de Sentencias 1 )
EL SIGUIENTE
MÁS
(Bloque de INSTRUCCIONES 2 )
TERMINARA SI

**3.7.1.- Sentencias condicionales.**

Sentencia de control donde la condición es la expresión que será evaluada. Si es verdadera el extracto se ejecuta. Si es falsa el extracto es ignorado. 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 la sentencia2.

EJEMPLO

```
#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

```
#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 / Operador 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; }
```

Mi Universidad

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

```
#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;
 }
}
```

**3.7.2.- Ciclos y bucles.**

Los ciclos o también conocidos como bucles, son una estructura de control esencial al momento de programar. Tanto C como C++ y la mayoría de los lenguajes utilizados actualmente, nos permiten hacer uso de estas estructuras. Un ciclo o bucle permite repetir una o varias instrucciones cuantas veces lo necesitemos, por ejemplo, si quisiéramos escribir los números del uno al cien no tendría sentido escribir cien líneas mostrando un numero en cada una, para esto y para muchísimas cosas más, es útil unciclo, permitiéndonos hacer una misma tarea en una cantidad de líneas muy pequeña y de forma prácticamente automática.

Existen diferentes tipos de ciclos o bucles, cada uno tiene una utilidad para casos específicos y depende de nuestra habilidad y conocimientos poder determinar en qué momento es bueno usar alguno de ellos. Tenemos entonces a nuestra disposición los siguientes tipos de ciclos en C++:

1. Ciclo for en C++
2. Ciclo while en C++
3. Ciclo do-while en C++

Como mencioné anteriormente, cada uno de estos ciclos tiene ciertas características que lo hacen útil para algunos casos específicos, a lo largo de los contenidos de esta sección veremos cada uno de estos al detalle, aprendiendo durante el proceso los componentes, sintaxis y esas características particulares que permiten decidir cual usar en qué momento, veremos también el concepto de contador y acumulador que parte de la existencia de los ciclos

Te recuerdo, que aunque intentaré profundizar bastante en cada concepto, lo haré enfocándome hacia el uso de los ciclos en el lenguaje C++ y no tanto hacia la parte de la lógica de los ciclos en particular, si tienes problemas con la correcta comprensión de la lógica y utilidad de cualquier tipo de ciclo o de los ciclos en general, te invito a que primero leas la sección de Fundamentación de los Ciclos, en dicha sección podrás comprender correctamente el funcionamiento de un ciclo y posteriormente podrás regresar a esta sección para aprender a implementar los ciclos en C++

Los ciclos for son lo que se conoce como estructuras de control de flujo cíclicas o simplemente estructuras cíclicas, estos ciclos, como su nombre lo sugiere, nos permiten ejecutar una o varias líneas de código de forma iterativa, conociendo un valor especifico inicial y otro valor final, además nos permiten determinar el tamaño del paso entre cada "giro" o iteración del ciclo.

En resumen, un ciclo for es una estructura de control iterativa, que nos permite ejecutar de manera repetitiva un bloque de instrucciones, conociendo previamente un valor deinicio, un tamaño de paso y un valor final para el ciclo.

Para comprender mejor el funcionamiento del ciclo for, pongamos un ejemplo, supongamos que queremos mostrar los números pares entre el 50 y el 100, si imaginamos un poco como

seria esto, podremos darnos cuenta que nuestro ciclo deberá mostrar una serie de números como la siguiente: 50 52 54 56 58 60 ... 96 98 100. Como podemos verificar, tenemos entonces los componentes necesarios para nuestro ciclo for, tenemos un valor inicial que sería el 50, tenemos también un valor final que sería el 100 y tenemos un tamaño de paso que es 2 (los números pares). Estamos ahora en capacidad de determinar los componentes esenciales para un ciclo for.

Vamos a ver ahora como es la sintaxis de un ciclo for en C++, así estaremos listos para Musarlos en nuestros programas de ahora en adelante

La sintaxis de un ciclo for es simple en C++, en realidad en la mayoría de los lenguajes de alto nivel es incluso muy similar, de hecho, con tan solo tener bien claros los 3 Mcomponentes del ciclo for (inicio, final y tamaño de paso) tenemos prácticamente todo hecho: for(int i = valor inicial; i <= valor final; i = i + paso)

```
{
 ....
 ....
 Bloque de Instrucciones....
 ....
 ....
}
```

Vamos ahora a ver línea por línea el anterior código para comprender todo y quedar claros. Posteriormente veremos un ejemplo con valores reales.

Línea 1:

En esta línea está prácticamente todo lo esencial de un ciclo for. La sintaxis es simple, tenemos una variable de control llamada i que es tipo entero (int), cabe notar que la variable se puede llamar como nosotros lo deseemos y puede ser del tipo de queramos también, sin embargo en la mayoría de los casos se usa la "i" como nombre y el entero como tipo, pero somos libres de modificar esto a nuestro gusto. Esta variable "i" se le asigna un valor inicial que puede ser cualquier número correspondiente al tipo de datoasignado. Posteriormente lo que haremos será especificar hasta donde irá nuestro ciclo por medio del valor final, ten en cuenta que cada uno de estos componentes es separado por un punto y coma ";", también es

importante saber que la condición final puede ser cualquier cosa, mayor, menor, mayor o igual, menor o igual, sin embargo no tiene sentido que la condición sea por ejemplo un igual, pues nuestra variable de control siempre va a cambiar entre valores, menores o mayores que el valor final deseado, si fuera un igual no tendríamos un error de sintaxis, pero nuestro for básicamente no haría nada de nada.

Finalmente el último componente de esta primer línea es el tamaño del paso, este componente se especifica aumentando en la cantidad deseada la variable de control.

Línea 2:

En la línea 2 tenemos una llave abriendo "{" lo cual como seguramente ya sabrás indica que allí comienza el bloque de instrucciones que se ejecutaran cada vez que el ciclo de un "giro". Esta llave no es del todo obligatoria, sin embargo si no la ponemos solo se ejecutara dentro de nuestro ciclo la primera línea inmediatamente posterior a la declaración del ciclo, de modo que si deseamos que se ejecuten varias líneas dentro de nuestro ciclo, debemos usar las llaves

Línea 3 a 7:

En estas líneas es donde estarán todas las operaciones que queramos llevar a cabo de manera iterativa durante la ejecución del ciclo, este bloque podrá tener la cantidad de líneas necesarias incluso, como veremos más adelante dentro de estas podría haber uno o más ciclos, así que podrías tener todo un programa dentro de un ciclo.

Línea 8:

En esta última línea hacemos uso de la llave cerrando "}", una vez más como seguramente ya sabrás esta nos indica que allí termina el bloque del ciclo for y se dará por terminada la ejecución de este para continuar ejecutando el resto del algoritmo.

No te preocupes si no comprendiste muy bien lo que acabo de escribir, estoy seguro que con un par de ejemplos que veremos a continuación, te va a quedar todo claro

Ejemplos de Ciclo For en C++

A continuación vamos a ver unos cuantos ejemplos para comprender de manera adecuada el uso de los ciclos for en c++, recuerda que si no comprendes alguno de estos ejemplos o tienes alguna pregunta o sugerencia sobre estos o cualquier contenido de la sección, puedes dejarlas en la sección de comentarios

Ejemplo 1: Mostrar en pantalla los números pares

Vamos a retomar el ejemplo anterior, donde deseábamos sacar los números pares entre el número 50 y el 100, es un ejemplo sencillo con el que nos aseguraremos de haber comprendido bien lo anterior:

Solución Ejemplo 1:

Como pudimos ver anteriormente, tenemos entonces que el valor inicial para nuestro ciclo es el numero 50 y el valor final es el 100, además, dado que necesitamos los números pares vamos a ir de dos en dos, así que el tamaño del paso va a ser 2, teniendo estos 3 componentes identificados, estamos listos para crear nuestro ciclo for así: for(int i=50;i<=100;i+=2)

```
{//Notemos que escribir i+=2 es similar a escribir i = i + 2
 cout << i << endl;
}
```

El código funcional completo sería el siguiente:

```
#include "iostream"
#include "stdlib.h"
using namespace std;
int main()
{
 for(int i=50;i<=100;i+=2)
 {//Notemos que escribir i+=2 es similar a escribir i = i + 2
 cout << i << endl;
 }
 system("PAUSE");
 return 0;
}
```

Los ciclos while son también una estructura cíclica, que nos permite ejecutar una o varias líneas de código de manera repetitiva sin necesidad de tener un valor inicial e incluso a veces sin siquiera conocer cuando se va a dar el valor final que esperamos, los ciclos while, no dependen directamente de valores numéricos, sino de valores booleanos, es decir su

ejecución depende del valor de verdad de una condición dada, verdadera o falso, nada más. De este modo los ciclos while, son mucho más efectivos para condiciones indeterminadas, que no conocemos cuando se van a dar a diferencia de los ciclos for, conlos cuales se debe tener claro un principio, un final y un tamaño de paso.

Para comprender mejor el funcionamiento del ciclo while, pongamos un buen ejemplo, imaginemos que por algún motivo, queremos pedirle a un usuario una serie de números cualquiera y que solo dejaremos de hacerlo cuando el usuario ingrese un número mayor a 100. Como vemos, aquí no podríamos utilizar un ciclo for, pues no tenemos ni idea de cuándo al usuario se le va a ocurrir ingresar un número mayor que 100, es algo indeterminado para nosotros, sin embargo el ciclo while nos permite ejecutar una acción de forma infinita hasta que se cumpla alguna condición especifica, en nuestro caso sería que el numero ingresado sea mayor a 100. De modo que si el usuario nos ingresa de manera sucesiva los siguientes numero 1,50,99, 49, 21, 30, 100 ..., nuestro programa no finalizara, pues ninguno de estos números es mayor que 100, sin embargo si nos ingresara el numero 300, el programa finalizaría inmediatamente.

Vamos a ver ahora como es la sintaxis de un ciclo while en C++, así estaremos listos para usarlos en nuestros programas de ahora en adelante cada vez que lo necesitemos.

La sintaxis de un ciclo while es incluso más simple y "legible" que la del ciclo for en C++, pues simplemente requerimos tener clara una condición de parada. En realidad, en la mayoría de los lenguajes de alto nivel la manera de escribir un ciclo while (la sintaxis) esincluso muy similar, así que con tan solo tener bien clara una condición de finalización para el ciclo tendremos prácticamente todo hecho. while(condición de finalización) //por ejemplo numero == 100

```
{
 ....
 ....
 Bloque de Instrucciones....
 ....
 ....
}
```

Vamos entonces a ver línea por línea el anterior código para comprender todo y quedar claros. Posteriormente veremos el ejemplo planteado anteriormente y su solución.

Línea 1:

En esta línea está prácticamente todo lo esencial de un ciclo while. La sintaxis es bastante simple. Tenemos al interior de los paréntesis una condición cualquiera, es decir por ejemplo "==. >, <, >=, <=, !=" o algunas mas que se nos puedan ocurrir, esta condición que especifiquemos allí, es la que permitirá que el ciclo se siga ejecutando hasta que en algún momento esta misma condición deje de cumplirse, de esta forma si por ejemplo estamos verificando que un numero_cualquiera == 50, el ciclo se ejecutara solo cuando numero_cualquiera sea igual a 50, en cuanto su valor cambie a cualquier otro el ciclo while finalizara y continuara con el resto de la ejecución del programa. De esta forma, es evidente que la condición que allí ingresemos siempre deberá tomar un valor booleano (true o false).

Línea 2:

En la línea 2 tenemos una llave abriendo "{" lo cual como sabemos indica que allí comienza un bloque de instrucciones que se ejecutaran cada vez que el ciclo de un "giro". Esta llave no es del todo obligatoria, sin embargo si no la ponemos solo se ejecutara dentro de nuestro ciclo while la primera línea inmediatamente posterior a la declaración del ciclo, de modo que si deseamos que se ejecuten varias líneas dentro de nuestro ciclo, debemos usar las llaves

Línea 3 a 7:

En estas líneas es donde estarán todas las operaciones que queramos llevar a cabo de manera iterativa durante la ejecución del ciclo, este bloque podrá tener la cantidad de líneas necesarias incluso, como veremos más adelante dentro de estas podría haber uno o más ciclos, así que podrías tener todo un programa dentro de un ciclo.

Línea 8:

En esta última línea hacemos uso de la llave cerrando "}", una vez más como seguramente ya debemos saber esta nos indica que allí termina el bloque del ciclo while y se dará por terminada la ejecución de este para continuar ejecutando el resto del algoritmo.

No te preocupes si no comprendiste muy bien lo que acabo de escribir, estoy seguro que con un par de ejemplos que veremos a continuación, te va a quedar todo claro.

Vamos a retomar el ejemplo anterior, donde queremos hacer que nuestro programa le pida a un usuario una serie de números cualquiera y que solo dejaremos de hacerlo cuando el usuario ingrese un número mayor a 100, una vez mas es un ejemplo sencillo con el que nos aseguraremos de haber comprendido bien todos los conceptos anteriores:

Solución Ejemplo 1:

Para solucionar esto, debemos tener clara cuál va a ser la condición que se debe cumplir para que el ciclo este pidiendo el numero contantemente, el ciclo se va a detener solo cuando el numero ingresado sea mayor que 100, así que la condición para que se siga ejecutando es que el numero sea menor a 100, ¿Comprender la lógica?, es simple si para que se detenga el numero debe ser mayor a 100, entonces para seguirse ejecutando el numero debe ser menor o igual a 100, veámoslo entonces

```
int numero;
cin >> numero;
while(numero <= 100)
{
 cout << "Ingrese un numero ";
 cin >> numero;
}
```

El código funcional completo y un tanto más amigable para el usuario sería el siguiente:

```
#include "iostream"
using namespace std;
int main()
{
 int numero;
 cout << "Ingrese un numero ";
 cin >> numero;
 while(numero <= 100)
 {
 cout << "Ingrese un numero ";
 cin >> numero;
```

```
 }
 system("PAUSE");
 return 0;
}
```

Los ciclos do-while son una estructura de control cíclica, los cuales nos permiten ejecutar una o varias líneas de código de forma repetitiva sin necesidad de tener un valor inicial e incluso a veces sin siquiera conocer cuando se va a dar el valor final, hasta aquí son similares a los ciclos while, sin embargo el ciclo do-while nos permite añadir cierta ventaja adicional y esta consiste que nos da la posibilidad de ejecutar primero el bloque de instrucciones antes de evaluar la condición necesaria, de este modo los ciclos do-while, son más efectivos para algunas situaciones especificas. En resumen un ciclo do-while, es una estructura de control cíclica que permite ejecutar de manera repetitiva un bloque de instrucciones sin evaluar de forma inmediata una condición especifica, sino evaluándola justo después de ejecutar por primera vez el bloque de instrucciones.

Para comprender mejor el funcionamiento del ciclo while, usemos de nuevo el ejemplo de la sección anterior sobre el ciclo while. Imaginemos entonces que por algún motivo, queremos pedirle a un usuario una serie de números cualquiera y que solo dejaremos de hacerlo cuando el usuario ingrese un número mayor a 100. Como vimos anteriormente, esto se puede hacer por medio de un ciclo while, pero vamos ahora a ver como lo podemos hacer usando un ciclo do-while mejorando así un poco nuestro algoritmo, evitando ciertos comandos, tal como se dijo con el ciclo while, en efecto aquí estamos en la situación de no tener ni idea de cuándo al usuario se le va a ocurrir ingresar un número mayor que 100, pues es algo indeterminado para nosotros, sin embargo el ciclo while y en efecto el do-while nos permite ejecutar cierta acción de forma infinita hasta que se cumpla alguna condición específica, en nuestro caso sería que el numero ingresado sea mayor a

100. De modo que si el usuario nos ingresa de manera sucesiva los siguientes número 1,50,99, 49, 21, 30, 100 ..., nuestro programa no finalizara, pues ninguno de estos números es mayor que 100, sin embargo si nos ingresara el número 300, el programa finalizaría inmediatamente.

Vamos a ver ahora como es la sintaxis de un ciclo do-while en C++, así estaremos listos para usarlos en nuestros programas de ahora en adelante cada vez que lo necesitemos.

La sintaxis de un ciclo do-while es un tanto más larga que la del ciclo while en C++, sin embargo no se hace más complicado, de hecho con tan solo tener bien clara una condición de finalización para el ciclo tendremos prácticamente todo terminado.

```
do
{
 ....
 ....
 Bloque de Instrucciones....
 ....
 ....
}
while(condición de finalización); //por ejemplo numero != 23
```

Vamos entonces a ver línea por línea el anterior código para comprender todo y quedar claros. Posteriormente veremos el ejemplo planteado anteriormente y su solución.

Línea 1:

Esta línea es por decirlo así, la parte novedosa del ciclo do-while, esta expresión no evalúa ninguna condición ni nada, simplemente da paso directo al bloque de instrucción y luego permite la evaluación de la condición.

Línea 2:

En la línea 2 tenemos una llave abriendo "{" lo cual como sabemos indica que allí comienza un bloque de instrucciones que se ejecutaran cada vez que el ciclo de un "giro". Esta llave no es del todo obligatoria, sin embargo si no la ponemos solo se ejecutará dentro de nuestro ciclo la primera línea inmediatamente posterior a la instrucción do, de modo que si deseamos que se ejecuten varias líneas dentro de nuestro ciclo, debemos usar las llaves.

En lo personal, es preferible poner siempre las llaves sin importar cuantas líneas se vayan a ejecutar, es una buena práctica de programación y te puede evitar dolores de cabeza

Línea 3 a 7:

En estas líneas es donde estarán todas las operaciones que queramos llevar a cabo de manera iterativa durante la ejecución del ciclo, este bloque podrá tener la cantidad de líneas necesarias incluso, como veremos más adelante dentro de estas podría haber uno o más ciclos, así que podrías tener todo un programa dentro de un ciclo.

Línea 8:

En esta última línea hacemos uso de la llave cerrando "}", una vez más como seguramente ya debemos saber esta nos indica que allí termina el bloque de instrucciones que se ejecutarán de manera cíclica y se dará por terminada la ejecución de este para continuar ejecutando el resto del algoritmo.

Línea 9:

La línea 9 en el ciclo do-while, tiene la misma importancia y función que l alinea 1 en la sección del ciclo while, cabe resaltar que simplemente evalúa la condición y define si se cumple o no para seguir con la ejecución del ciclo o con la del resto del algoritmo, de este modo podemos ver que el ciclo while y el do-while son muy similares, con la pequeña diferencia en que en uno se evalúa la condición desde el principio y en la otra al final de cada ciclo.

No te preocupes si no comprendiste muy bien lo que acabo de escribir, estoy seguro que con un par de ejemplos que veremos a continuación, te va a quedar todo claro.

A continuación vamos a ver unos cuantos ejemplos para comprender de manera adecuada el uso de los ciclos while en c++.

Vamos a retomar el ejemplo anterior, donde queremos hacer que nuestro programa le pida a un usuario una serie de números cualquiera y que solo dejaremos de hacerlo cuando el usuario ingrese un número mayor a 100, una vez mas es un ejemplo sencillo con el que nos aseguraremos de haber comprendido bien todos los conceptos anteriores, vamos a ver cómo hacer lo mismo con dos tipos de ciclos diferentes (el while y el dowhile), sin embargo vamos a ver como con uno es más eficiente que con el otro:

Solución Ejemplo 1:

Para solucionar esto, debemos tener clara cuál va a ser la condición que se debe cumplir para que el ciclo este pidiendo el número constantemente. El ciclo se va a detener solo cuando el número ingresado sea mayor que 100, así que la condición para que se siga ejecutando es

que el numero sea menor a 100, ¿Comprender la lógica?, es simple si para que se detenga el numero debe ser mayor a 100, entonces para seguirse ejecutando el numero debe ser menor o igual a 100, veámoslo entonces

```
int numero;
do
{
 cout << "Ingrese un numero ";
 cin >> numero;
}
while(numero <= 100);
```

El código funcional completo y un tanto más amigable para el usuario sería el siguiente:

```
#include "iostream"
using namespace std;
int main()
{
 int numero; do
 {
 cout << "Ingrese un numero ";
 cin >> numero;
}
 while(numero <= 100);
 system("PAUSE");
 return 0;
}
```

### 3.7.2.- Etiquetas y GOTO.

La instrucción de salto goto se puede usar en un programa, para transferir m incondicionalmente el control del mismo a la primera instrucción después de una etiqueta, o dicho de otra forma, al ejecutar una instrucción goto, el control del programa se transfiere (salta) a la primera instrucción después de una etiqueta. Una etiqueta se define mediante su nombre (identificador) seguido del carácter dos puntos (:).

Ejemplo:

```
#include <stdio.h>
int main()
{
 int n, a;
 a = 0;
 do
 {
 printf( "Introduzca un numero entero: " );
 scanf( "%d", &n );
 if ( n == 0 )
 {
 printf( "ERROR: El cero no tiene opuesto.\n" );
 goto etiqueta_1;
 /* En el caso de que n sea un cero, el control de programa salta a la primera instrucción después de etiqueta_1. */
 }
 printf( "El opuesto es: %d\n", -n );
 a += n;
 } while ( n >= -10 && n <= 10 );
 etiqueta_1:
 printf( "Suma: %d", a );
 return 0;
}
```

En pantalla:

```
 Introduzca un número entero: -4
 El opuesto es: 4
Introduzca un número entero: 12
 El opuesto es: -12
 Introduzca un número entero: 0
```

ERROR: El cero no tiene opuesto.

Suma: 8

3.8.- Tipos de datos estructurados.

C permite definir estructuras de datos que agrupan campos de otros tipos de datos. La sintaxis se muestra a continuación:

```
struct nombre_de_la_estructura
{
 tipo_1 nombre_del_campo1;
 tipo_2 nombre_del_campo2;
 ...
 tipo_N nombre_del_campoN;
};
```

La construcción anterior sólo define un nuevo tipo de datos, no se declara variable alguna. Es decir, la construcción anterior tiene la misma entidad que el tipo "int" o "float". El nombre del nuevo tipo estructurado definido es "struct nombre_de_la_estructura". Por ejemplo:

```
#define FIRST_SIZE 100
#define LAST_SIZE 200
#define CONTACTS_NUM 100
/* Definición de la estructura */
struct contact_information
{
 char firstname[FIRST_SIZE];
 char lastname[LAST_SIZE];
 unsigned int homephone;
 unsigned int mobilephone;
};
/* Declaración de variables con esta estructura */
struct contact_information person1, person2, contacts[CONTACTS_NUM];
```

Las líneas 6 a 12 definen un nuevo tipo de datos estructurado que contiene cuatro campos, los dos primeros son tablas de letras y los dos últimos son enteros. A pesar de que estos

campos tienen nombres y tamaños, hasta el momento no se ha declarado ninguna variable. Es en la línea 15 en la que se sí se declaran tres variables de este nuevo tipo estructurado. La última de ellas es una tabla de 100 de estas estructuras. Asegúrate de que tienes clara la diferencia entre la "definición" de un tipo de datos y la "declaración" de variables de ese tipo. La siguiente figura muestra estos conceptos para una estructura y un tipo básico.

2.2. Tipos de datos estructurados

| | Tipo de datos básico | Tipo de datos estructurado |
|---|---|---|
| Definición | int | ruct contact_information {<br>char firstname[100];<br>char lastname[200];<br>int homephone;<br>int mobilephone;<br>}; |
| Declaración | int a, b[100]; | struct contact_information a, b[100]; |

3.8.1.- Arrays.

Para crear un array de longitud variable se pueden utilizar punteros. El array se declara así:

int *array;

Cuando queramos reservar espacio para él

/* En C++ */

array = new int [N];

/* En C o en C++ */

array = (int *)malloc (N*sizeof(int)); donde N es el número de elementos que queremos para el array. En C++ nos basta con indicar en el new el número de elementos. Con C debemos indicar en la fución malloc() el tamaño en bytes que deseamos para el array. Si sizeof(int) es lo que ocupa un entero, necesitamos N*sizeof(int)

Para usar el array, una vez creado, simplemente

array[i] = ...; variable = array[i];

Una vez que terminemos de usarlo, debemos liberarlo

/* Si hemos hecho new */

delete [ ] array;

/* Si hemos hecho malloc */ free(array);

Para un array en dos dimensiones, debemos hacer una declaración más compleja. Un array de dos dimensiones es en realidad un array de una dimensión de punteros. Cada uno de

estos punteros apuntará a otro array de una dimensión de los datos que queremos. Cada uno de estos arrays será una fila de nuestro array de dos dimensiones. La declaración, por tanto, es compleja

```
int **array;
```

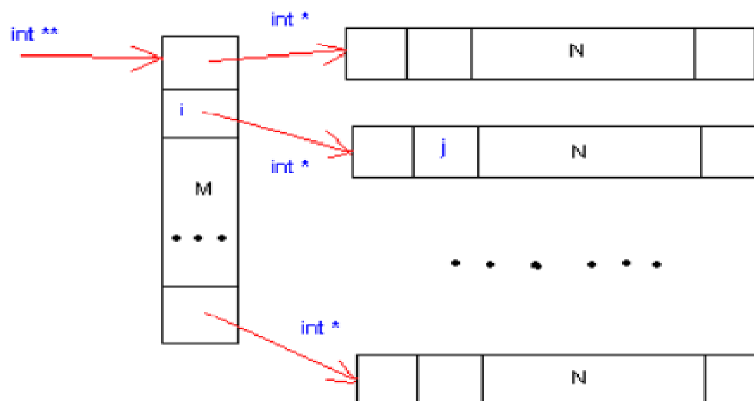
Para reservar espacio, por ejemplo M filas y N columnas, debemos primero reservar espacio para el array de punteros

```
/* En C++ */
array = new (int *)[M];
/* En C */
array = (int **)malloc (M * sizeof(int *));
```

Es decir, hacemos un array de M punteros a entero, es decir, de M int *. Ya tenemos creados tantos punteros como filas, pero no las columnas. Ahora hay que crear las columnas. Las columnas se crean haciendo un array de una dimensión con tantos elementos como columnas. Para ello, debemos hacer un bucle para cada fila en el que creemos un nuevo array

```
/* En C++ */
for (int i=0; i<M; i++)
 array[i] = new int[N];
/* En C */
int i;
for (i=0; i<M; i++)
 array[i] = (int *)malloc (N*sizeof(int));
```

Al final hemos construido algo como lo de la figura



Para usarlo, simplemente usamos dos corchetes

array[i][j] = valor; valor = array[i][j];

Con array[i] tenemos el puntero i que apunta a un array de una dimensión, quecorresponde a la fila i. Luego con array[i][j] accedemos al elemento j de ese array[i].

Liberar esto puede ser un poco pesado. Primero debemos liberar cada uno de los arrays fila y luego el de punteros.

```
/* En C++ */
for (int i=0; i<M; i++)
 delete [] array[i];
delete [] array;
/* En C */
int i;
for (i=0; i<M; i++) free (array[i]); free(array);
```

**3.8.2.- Estructuras.**

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.

```
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 m 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}};
```

3.8.3.- Uniones.

Podemos enviar una estructura a una unión de las dos maneras conocidas:

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);
}
```

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);
}
```

**3.8.4.- Tipos de enumerados.**

Las enumeraciones consisten en la palabra clave enum y un identificador opcional seguido de una lista de enumeradores entre llaves.

Un identificador es de tipo int.

La lista de enumeradores tiene al menos un elemento de enumerador.

A un enumerador se le puede "asignar" opcionalmente una expresión constante de tipo int.

Un enumerador es constante y es compatible con un char , un entero con signo o un entero sin signo. Lo que se usa siempre está definido por la implementación. En cualquier caso, el tipo utilizado debe poder representar todos los valores definidos para la enumeración en cuestión.

Si no se "asigna" una expresión constante a un enumerador y es la primera entrada en una lista de enumeradores, toma el valor de 0 , de lo contrario, toma el valor de la entrada anterior en la lista de enumeradores más 1.

El uso de múltiples "asignaciones" puede llevar a que diferentes enumeradores de la misma enumeración tengan los mismos valores.

Una enumeración simple es un tipo de datos definido por el usuario que consta de constantes integrales y cada constante integral recibe un nombre. La enum palabras clave se utiliza para definir el tipo de datos enumerados.

Si usa enum lugar de int o string/ char* , aumenta la verificación en tiempo de compilacióny evita que los errores pasen en constantes no válidas, y documenta qué valores son legales usar.

Ejemplo 1

```
#
enum color{ RED, GREEN, BLUE };
void printColor(enum color chosenColor)
{
 const char *color_name = "Invalid color";
 switch (chosenColor)
 {
```

```
 case RED:
 color_name = "RED";
 break;
 case GREEN:
 color_name = "GREEN";
 break;
 case BLUE:
 color_name = "BLUE";
 break;
 }
 printf("%s\n", color_name);
}
```

Con una función principal definida como sigue (por ejemplo):

```
int main(){
 enum color chosenColor;
 printf("Enter a number between 0 and 2");
 scanf("%d", (int*)&chosenColor);
 printColor(chosenColor);
 return 0;
}
```

Enumeración de tipo topográfico.- Los tipos de enumeración también se pueden declarar sin darles un nombre:

```
 enum { buffersize = 256, };
 static unsigned char buffer [buffersize] = { 0 };
```

Esto nos permite definir constantes de tiempo de compilación de tipo int que, como en este ejemplo, se pueden usar como longitud de matriz.

Enumeración con valor duplicado.- Un valor de enumeración de ninguna manera debe ser único:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
```

```
enum Dupes
{

 Base, /* Takes 0 */
 One, /* Takes Base + 1 */
 Two, /* Takes One + 1 */
 Negative = -1,
 AnotherZero /* Takes Negative + 1 == 0, sigh */
};

int main(void)
{
 printf("Base = %d\n", Base);
 printf("One = %d\n", One);
 printf("Two = %d\n", Two);
 printf("Negative = %d\n", Negative);
 printf("AnotherZero = %d\n", AnotherZero);
 return EXIT_SUCCESS;

}
```

La muestra se imprime:

```
Base = 0
One = 1
Two = 2
Negative = -1
AnotherZero = 0
```

**UNIDAD IV**

**PUNTEROS**

### 4.1.- Definición de punteros.

Un puntero es una variable que contiene la dirección de otra variable. También podríamos decir que un puntero es una variable que representa la posición (más que el valor) de otro dato, tal como una variable o un elemento de un array. Cuando una variable puntero es definida, el nombre de la variable debe ir precedido de un asterisco (*). Este identifica que la variable es un puntero. Por tanto, una declaración de puntero puede ser escrita en términos generales como: <tipo> es cualquier tipo de variable en C. <identificador> es el nombre del puntero. El tipo o tipo base, indica el tipo de variables que se podrán manipular a través del puntero. Es importante conocer el tipo base de un puntero, puesto que toda la aritmética de punteros se realiza con relación a la base.

(Luego lo veremos más claro en los ejemplos).

- & Devuelve la dirección de memoria del operando.
- * Devuelve el valor almacenado en la dirección de memoria que determina el operando.

Los punteros se pueden comparar entre ellos y se pueden asignar direcciones de memoria. Además, se pueden decrementar o incrementar. El incremento o el decremento varían según el tipo de dato al que apunten. A los punteros también se les puede sumar o restar números enteros. NO SE PUEDE REALIZAR NINGUNA OTRA OPERACIÓN ARITMÉTICA CON LOS PUNTEROS.

Los punteros son usados con frecuencia en C, y tienen gran cantidad de aplicaciones:

- Proporcionan una forma de devolver varios datos desde una función mediante los argumentos de la función.
- Nos permiten igualmente, que referencias a otras funciones puedan ser especificadas como argumentos de una función (pasar funciones como argumentos en una función determinada).

Supongamos que v una variable que representa una determinado dato, esta variable le corresponde una dirección de memoria, esta dirección de memoria puede ser accedida mediante &v (el operador unario & proporciona la dirección del operando v). Es importante

recordar que si asignamos &v a una variable pv, dicha variable me va a representar la dirección de memoria de v, y no su valor. Si queremos acceder al valor que posee la dirección de x tendremos que poner: *pv (donde * es un operador unario, llamado operador indirección, que opera sólo sobre una variable puntero). Entonces vemos que tanto v como como *pv, representan el mismo valor. Si ahora asignáramos a una variable u el valor de *pv, entonces voy a tener u y v representando al mismo valor.

### 4.2.- Operación con punteros.

Un puntero es un tipo de dato similar a un entero, y hay un conjunto de operaciones definidas para punteros:

- La suma o resta de un entero produce una nueva localización de memoria.
- Se pueden comparar punteros, utilizando expresiones lógicas, para ver si están apuntando o no a la misma dirección de memoria.
- La resta de dos punteros da como resultado el número de variables entre las dos direcciones.

Veamos un ejemplo de utilización de punteros:

```
#include <iostream.h>
main()
{
 int vector[3];
 int* princPunt = vector;
 int* finPunt = &vector[2]; vector[2] = 15;
 cout << *(princPunt+2) << '\t' << *finPunt <<'\n';
 if (princPunt == finPunt)
 cout << " Esto no puede suceder " << '\n';
 cout << "Numero de elementos \t" <<finPunt-princPunt << '\n';
}
```

El resultado de la ejecución de este programa es:

15 15

Número de elementos 2

Veamos cómo trabaja este programa:

- princPunt es la dirección del primer elemento de vector, y finPunt la dirección del último elemento. int princPunt = vector; es una combinación de declaración y definición.
- La expresión *(princPunt+2) incrementa el valor del puntero princPunt en dos y devuelve el número guardado en esa localización, es decir, apunta a la tercera localización de memoria y su valor es 15.
- Se pueden utilizar también enteros en formato hexadecimal. Así, cout << *(princPunt + 17) y cout << *(princPunt + 0x11 ) producen la misma salida.
- La expresión princPunt == finPunt comprueba si los dos punteros son iguales. Estosólo puede ser verdad si los dos punteros apuntan a la misma variable.

Siempre que se realiza una operación aritmética sobre un puntero, sumando o restando un entero, el puntero se incrementa o decrementa un número apropiado de sitios tal que el nuevo valor apunta a la variable que está n elementos (no n bytes) antes o después que el dado. De la misma forma, al restar dos punteros se obtiene el número de objetos entre las dos localizaciones. Finalmente, dos punteros son iguales si y sólo si apuntan a la misma variable (el valor de las direcciones es el mismo). No son necesariamente iguales si sus valores indirectos son los mismos, ya que estas variables podrían estar en diferentes localizaciones de memoria.

La siguiente tabla resume los operadores que manipulan punteros:

| Operador | Acción |
|---|---|
| *p | Contenido |
| &x | Puntero a un objeto |
| p[i] | Elemento de un vector |
| p->m | Selección de miembros de una clase |
| ++p | Preincremento al siguiente elemento |
| p++ | Postincremento al siguiente elemento |
| --p | Predecremento al elemento anterior |
| p-- | Postdecremento al elemento anterior |
| p += n | Incremento en n elementos |
| p -= n | Decremento en n elementos |
| p+n | Offset en n elementos |
| p-n | Offset en n elementos |
| new T | Reserva de memoria para el objeto T |
| new T[n] | Reserva memoria para un vector de n objetos de tipo T |
| delete p | Libera la memoria reservada para p |
| delete [] p | Libera la memoria reservada para un vector de objetos |

**4.3.- Punteros y Arrays.**

En C hay muy poca diferencia "interna" entre un puntero y un array. En muchas ocasiones, podremos declarar un dato como array (una tabla con varios elementos iguales, de tamaño predefinido) y recorrerlo usando punteros. Vamos a ver un ejemplo:

```
#include <stdio.h>
int main() {
int datos[10];
 int i;
 /* Damos valores normalmente */
 for (i=0; i<10; i++) datos[i] = i*2;
 /* Pero los recorremos usando punteros */
 for (i=0; i<10; i++)
 printf ("%d ", *(datos+i));
 return 0;
}
```

Pero también podremos hacer lo contrario: declarar de forma dinámica una variable usando "malloc" y recorrerla como si fuera un array:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
 int *datos;
 int i;
 /* Reservamos espacio */ datos = (int *) malloc (20*sizeof(int));
 /* Damos valores como puntero */
 printf("Uso como puntero... ");
 for (i=0; i<20; i++)
 *(datos+i) = i*2;
 /* Y los mostramos */
 for (i=0; i<20; i++)
 printf ("%d ", *(datos+i));
```

```
 /* Ahora damos valores como array */
 printf("\nUso como array... ");
 for (i=0; i<20; i++) datos[i] = i*3;
 /* Y los mostramos */
 for (i=0; i<20; i++)
 printf ("%d ", datos[i]);
 /* Liberamos el espacio */ free(datos);
80
 return 0;
}
```

**4.4.- Punteros y funciones.**

Anteriormente vimos que las direcciones de memoria obtenidas con el operador & nos sirven para utilizar funciones predeterminadas de C, y específicamente describimos cómo utilizar la función scanf, que toma estas direcciones como argumentos que le indican dónde guardar los datos leídos. Ahora veremos una segunda aplicación de las direcciones de memoria: cómo y cuándo crear nuestras propias funciones que trabajen con direcciones de memoria.

Ejemplo

Consideremos el siguiente programa en C:

```
{incrementar_v1.c 2}
#include <stdio.h>
void incrementar(int n);

int main () {
 int contador = 0;
 printf("[main] Contador antes de incrementar: %d\n", contador); incrementar(contador);
 printf("[main] Contador despues de incrementar: %d\n", contador);
 return 0;
}
void incrementar (int n) {
```

```
 n = n + 1;
}
```

¿Qué es el tipo de retorno void? Básicamente, el tipo de retorno void indica que la función no retorna un valor. ¿Por qué queremos llamar a una función que no retorna nada? Pues, por los efectos computacionales que produce: en este caso incrementar una variable.

¿Qué es lo que se imprimirá por pantalla debido a la ejecución de este programa?

Lo que se mostrará por pantalla es:

[main] Contador antes de incrementar: 0

[main] Contador despues de incrementar: 0

¿Coincide este resultado con la intención aparente de este programa? Pues claro que no, ya que basado en los nombres de las variables y de las funciones involucradas, se esperaría que la función incrementar de alguna manera haga que el contador aumente en una unidad. Sin embargo esto no sucede... Veamos qué pasa dentro de la función incrementar.

{incrementar_v2.c 2}

```
#include <stdio.h>
void incrementar(int n);
int main () {
int contador = 0;
 printf("[main] Contador antes de incrementar: %d\n", contador); incrementar(contador);
 printf("[main] Contador despues de incrementar: %d\n", contador);
 return 0;
}
void incrementar (int n) {
 printf("[incrementar] n antes de sumar: %d\n", n);
 n = n + 1;
 printf("[incrementar] n despues de sumar: %d\n", n);
}
```

Al ejecutar este programa tenemos la siguiente salida por pantalla:

[main] Contador antes de incrementar: 0

[incrementar] n antes de sumar: 0

[incrementar] n despues de sumar: 1

[main] Contador despues de incrementar: 0

¿Qué es lo que está pasando? Los cambios al parámetro de la función solo son "visibles" o efectivos dentro de la función. Pero una vez fuera, la variable original que se pasó como argumento no se ve modificada. Esto se conoce como paso de parámetros por valor y también como paso de parámetros por copia.

Siguiendo nuestra intuición y en base a lo que hemos visto hasta ahora, podemos pensar que al igual que con la función scanf lo que debemos hacer es usar como argumento la dirección de memoria de la variable contador, es decir:

{incrementar_v3.c 2}

```
#include <stdio.h>
void incrementar(int n);
82
int main () {
 int contador = 0;
 printf("[main] Contador antes de incrementar: %d\n", contador); incrementar(&contador); //
**** Llamada usando direccion de memoria!
 printf("[main] Contador despues de incrementar: %d\n", contador);
 return 0;
}
void incrementar (int n) {
 printf("[incrementar] n antes de sumar: %d\n", n);
 n = n + 1;
 printf("[incrementar] n despues de sumar: %d\n", n);
}
```

Pero, al tratar de compilar este programa obtenemos un mensaje de advertencia:

warning: incompatible pointer to integer conversion passing 'int *' to parameter of type 'int'; remove & [-Wint-conversion] incrementar(&contador);

¿Qué significa esta advertencia? Aquí la parte clave es cuando dice:

passing int * to parameter of type int

Esto quiere decir que la función incrementar está esperando un parámetro de tipo int, algo que podemos ver claramente en su prototipo e implementación, pero que nuestra llamada incrementar (&contador) estamos pasando un argumento de tipo int * ¿qué significa esto?

Punteros como Nuevos Tipos de Datos

Volviendo nuevamente a lo que ya sabemos, dijimos que usaríamos &contador porque queremos pasar la dirección de memoria de la variable contadora la función incrementar para que efectivamente el cambio pueda ser "visto" después de ejecutada la función. Por lo tanto, &contador es efectivamente un valor, cuyo tipo de dato es int * y que contiene la dirección de memoria de la variable contador. En otras palabras:

Puntero a entero: Un valor de tipo int * es aquel que contiene la dirección de memoria a una variable de tipo int. Un valor de tipo int * se conoce como un puntero a un entero.

¿Qué pasa con los otros tipos de datos? Bueno: un puntero a un valor de tipo float tiene tipo float * un puntero a un valor de tipo double tiene tipo double * si defino un nuevo tipo de dato Fecha, usando typedef y struct, tambien tengo punteros a fecha, con tipo Fecha *. y similarmente para los otros tipos de datos...

Entonces, podemos concluir con la siguiente definición:

Puntero: un puntero es un tipo de dato que representa las direcciones de memoria que apuntan a una variable específica. Cada puntero apunta a un tipo de dato específico. O al revés, cada tipo de dato tiene un puntero como tipo de dato asociado. Cuando hablamos de puntero nos referimos a cualquier puntero en general. Pero en el código, siempre debemos hablar de puntero a..., y especificar a qué tipo de dato está asociada la dirección de memoria almacenada en el puntero.

Volvamos a nuestro ejemplo: ahora sabemos que en realidad para que la función incrementar pueda cambiar el valor del parámetro, ésta debe recibir como argumento no un valor de tipo int, sino que un puntero a un entero, es decir, un valor de tipo int *:

{incrementar_v4.c 3}

```
#include <stdio.h>
void incrementar(int *n);
```

```
int main () {
 int contador = 0;
 printf("[main] Contador antes de incrementar: %d\n", contador); incrementar(&contador); //
**** Llamada usando direccion de memoria!
 printf("[main] Contador despues de incrementar: %d\n", contador);
 return 0;
}
void incrementar (int *n) {
 printf("[incrementar] n antes de sumar: %d\n", n);
 n = n + 1;
 printf("[incrementar] n despues de sumar: %d\n", n);
%}
```

¿Qué es lo que pasa ahora? Al compilar obtenemos dos nuevos warnings:

warning: format specifies type 'int' but the argument has type 'int *' [-Wformat]

printf("[incrementar] n antes de sumar: %d\n", n); y

warning: format specifies type 'int' but the argument has type 'int *' [-Wformat]

printf("[incrementar] n despues de sumar: %d\n", n); lo que en realidad son dos instancias del mismo problema: en nuestro uso de printf estamos diciendo que imprimeros un entero, al especificar el formato %d pero en realidad estamos imprimiento un puntero a un entero. Por otro lado, si ejecutamos nuestro programa, obtendremos una salida como esta:

[main] Contador antes de incrementar: 0 [incrementar] n antes de sumar: 1466738584 [incrementar] n despues de sumar: 1466738588

[main] Contador despues de incrementar: 0

Todavía sigue sin funcionar como queremos!! Y ahora además vemos que los valores impresos para n son claramente incorrectos. Esto ocurre por que estamos usando un especificador de formato incorrecto para printf. De todos modos es importante observar, que el valor de n sí fue incrementado.

## 4.5.- Asignación dinámica de memoria.

Otra de las grandes ventajas de la utilización de punteros es la posibilidad de realizar una asignación dinámica de memoria. Esto significa que la reserva de memoria se realiza dinámicamente en tiempo de ejecución, no siendo necesario entonces tener que especificar en la declaración de variables la cantidad de memoria que se va a requerir. La reserva de memoria dinámica añade una gran flexibilidad a los programas porque permite al programador la posibilidad de reservar la cantidad de memoria exacta en el preciso instante en el que se necesite, sin tener que realizar una reserva por exceso en prevención a la que pueda llegar a necesitar.

Dado que los punteros se pueden aplicar a cualquier tipo de variable se puede entonces realizar una asignación de memoria dinámica para cualquier variable. La función malloc es la que se utiliza para realizar una reserva de memoria y se encuentra en el archivo de cabecera <stdlib.h>. La cabecera de dicha función es como sigue:

Ejemplo:Reserva de memoria de un dato simple:

```
#include<stdlib.h>
main() {
int *dato_simple;
dato_simple = (int *) malloc (1*sizeof(int));
}
```

Este trozo de código me reserva memoria para 1 dato int.

El argumento de la función malloc especifica el número de bytes de memoria que el usuario quiere reservar y devuelve la dirección de memoria de la zona de memoria reservada. Con objeto de facilitar el "cálculo del número de bytes necesarios", el usuario puede especificar dicho número en función del tipo de dato que quiere resevar y del número de datos de dicho tipo. De este modo, la sintaxis de la función malloc quedaría:

- num_elementos hace referencia al número de datos que se quiere resevar.
- malloc y sizeof son dos palabras reservadas de C. sizeof.
- tipo_dato se refiere al tipo de dato que se quiere reservar.

Ejemplo:Reserva de memoria de varios datos simples:

```
#include<stdlib.h>
```

```
main() {
int *dato_simple;
dato_simple = (int *) malloc (3*sizeof(int));
}
```

Este trozo de código me reserva memoria para 3 datos int.

La función malloc reserva una cantidad de memoria suficiente para almacenar num_elementos del tipo tipo_dato y devuelve la dirección de memoria de la celda del primer elemento reservado.

**4.6.- Entrada, salida y archivo de disco.**

El proceso de lectura de un archivo de texto es similar a la lectura desde el dispositivo estándar. Creamos un objeto entrada de la clase FileReader en vez de InputStreamReader.

El final del archivo viene dado cuando la función read devuelve -1. El resto del código es similar.

```
FileReader entrada=null;
StringBuffer str=new StringBuffer(); try { entrada=new FileReader("ArchivoApp2.java");
int c;
while((c=entrada.read())!=-1){
str.append((char)c);
}
} Catch (IOException ex) {}
```

Para mostrar el archivo de texto en la pantalla del monitor, se imprime el contenido del objeto str de la clase StringBuffer.

```
System.out.println(str);
```

Una vez concluído el proceso de lectura, es conveniente cerrar el flujo de datos, esto se realiza en una claúsula finally que siempre se llama independientemente de que se produzcan o no errores en el proceso de lectura/escritura.

```
}finally{
if(entrada!=null){ try{
entrada.close();
} Catch(IOException ex){}
```

```
 }
 }
```

El código completo de este ejemplo es el siguiente:

```
public class ArchivoApp2 {
 public static void main(String[] args) {
 FileReader entrada=null;
 StringBuffer str=new StringBuffer(); try { entrada=new FileReader("ArchivoApp2.java");
 int c;
 while((c=entrada.read())!=-1){
 str.append((char)c);
 }
 System.out.println(str);
 System.out.println("--------------------------------------");
 } Catch (IOException ex) {
 System.out.println(ex);
 }finally{
//cerrar los flujos de datos
 if(entrada!=null){ try{
 entrada.close();
 } Catch (IOException ex){}
 }
 System.out.println("el bloque finally siempre se ejecuta");
 }
 }
}
```

Los pasos para leer y escribir en disco son los siguientes:

Se crean dos objetos de las clases FileReader y FileWriter, llamando a los respectivos constructores a los que se les pasa los nombres de los archivos o bien, objetos de la clase File, respectivamente entrada=new FileReader("ArchivoApp3.java"); salida=new FileWriter ("copia.java");

Se lee mediante read los caracteres del flujo de entrada, hasta llegar al final (la función read devuelve entonces -1), y se escribe dichos caracteres en el flujo de salida mediante write.

```
 while((c=entrada.read())!=-1){
 salida.write(c);
 }
```

Finalmente, se cierran ambos flujos llamando a sus respectivas funciones close en bloques try..catch

```
 entrada.close();
 salida.close();
```

El código completo de este ejemplo que crea un archivo copia del original, es el siguiente

```
import java.io.*;
public class ArchivoApp3 {
 public static void main(String[] args) {
 FileReader entrada=null;
 FileWriter salida=null; try { entrada=new FileReader("ArchivoApp3.java"); salida=new FileWriter("copia.java");
 int c;
 while((c=entrada.read())!=-1){
 salida.write(c);
 }
 } Catch (IOException ex) {
 System.out.println(ex);
 }finally{
//cerrar los flujos de datos
 if(entrada!=null){ try{
 entrada.close();
 } Catch(IOException ex){}
 }
 if(salida!=null){ try{
```

```
salida.close();
} Catch(IOException ex){}
}
System.out.println("el bloque finally siempre se ejecuta");
}
}
}
```

### 4.6.1.- Flujos y archivos.

C ve cada uno de los archivos como un flujo secuencial de bytes. Cada archivo, termina con un marcador de fin de archivo o en un número de bytes específico registrado en una estructura administrativa de datos, mantenida por el sistema. Cuando un archivo se abre, se asocia un flujo con el archivo.

Al empezar la ejecución de un programa automáticamente se abren tres archivos y sus flujos asociados, la entrada estándar, la salida estándar y el error estándar. Los flujos proporcionan canales de comunicación entre archivos y programas. Así, el flujo de entrada estándar permite que un programa lea datos del teclado, el flujo de salida estándar permite que un programa imprima datos a la pantalla.

Abrir un archivo regresa un apuntador a una estructura FILE (definida en <stdio.h>) que contiene información utilizada para procesar dicho archivo, es decir, un índice a un arreglo del sistema operativo, conocido como una tabla de archivo abierto. Cada elemento contiene un bloque de control de archivo utilizado por el sistema operativo para administrar el archivo particular. La entrada estándar, la salida estándar y el error estándar son manejados utilizando los apuntadores de archivo stdin, stdout, stderr.

La biblioteca estándar proporciona muchas funciones para leer datos de los archivos y para escribir datos en los archivos.

### 4.6.2.- E/ S por consola: GETCHE( ) Y PUTCHAR( ).

En la biblioteca estándar de C se definen las dos principales vías de comunicación de un programa:

- la entrada estándar (stdin), y
- la salida estándar (stdout).

Generalmente están ambas asociadas a nuestro terminal, de manera que cuando se imprimen datos en la salida estándar los caracteres aparecen en el terminal, y cuando leemos caracteres de la entrada estándar los leemos del teclado del terminal. La entrada y salida estándar trabajan con caracteres (en modo carácter), con datos o números binarios.

Es decir, todos los datos que enviemos a la salida estándar deben ser cadenas de caracteres. Por ello para imprimir cualquier dato en la salida estándar primero deberemos convertirlo en texto, es decir, en cadenas de caracteres. Sin embargo esto lo haremos mediante las funciones de biblioteca, que se encargan de realizar esta tarea eficientemente: putchar y getchar

Comenzaremos con las dos funciones principales de salida y entrada de caracteres: putchar y getchar.

La función putchar escribe un único carácter en la salida estándar. Su uso en sencillo y generalmente está implementada como una macro en la cabecera de la biblioteca estándar.

Por ejemplo:

```
#include <stdio.h>
main()
{
putchar('H');
putchar('o');
putchar('l');
putchar('a');
putchar(32);
89
putchar('m');
putchar('u');
putchar('n');
putchar('d');
putchar('o');
putchar('\n');
}
```

El resultado es:

Hola mundo

En el código anterior putchar(32); muestra el espacio entre ambas palabras (32 es el código ASCII del carácter espacio ' ') y putchar('\n'); imprime un salto de línea tras el texto.

La función getchar devuelve el carácter que se halle en la entrada estándar. Esta función tiene dos particularidades. La primera es que aunque se utiliza para obtener caracteres no devuelve un carácter, sino un entero. Esto se hace así ya que con un entero podemos representar tanto el conjunto de caracteres que cabe en el tipo carácter (normalmente el conjunto ASCII de caracteres) como el carácter EOF de fin de fichero. En UNIX es habitual representar los caracteres usando el código ASCII, tanto en su versión de 7 bits como en su versión ampliada a 8 bits. Estos caracteres se suelen representar como un entero que va del 0 al 127 o 256. El carácter EOF entonces es representado con un -1.

Además esto también lo aplicaremos cuando leamos los ficheros binarios byte a byte. Un ejemplo:

```
#include <stdio.h>
main()
{
int c;
c = getchar(); /* Nótese que getchar() no devuelve nada hasta que se presiona ENTER */
putchar(c);
}
```

Aquí se almacena en el entero c el carácter pulsado en el teclado. Posteriormente se muestra con putchar.

### 4.6.3.- E/S por consola con formato PRINTF ( ) Y SCANT( ).

printf: impresión en pantalla

Para imprimir datos de un modo más general el C dispone de la función printf, que se ocupa de la impresión con formato en la salida estándar. Cuando se empieza con un nuevo lenguaje suele gustar el ver los resultados, ver que nuestro programa hace "algo". Por eso la mayor parte de programas de principiantes utilizan rápidamente la función printf, que sirve para sacar información por pantalla.

Para utilizar la función printf en nuestros programas debemos incluir la directiva:

#include <stdio.h> al principio de programa, como se ha visto en el primer programa de la sección Estructura de un programa en C. Esto es porque esta función se halla definida en el archivo de cabecera stdio.h de la siguiente manera:

```
int printf ( const char *format [, argumentos, ...] );
```

Si sólo queremos imprimir una cadena basta con hacer:

```
printf( "Cadena" );
```

Esto dará como resultado en la pantalla:

Cadena

Lo que se ponga entre las comillas es lo que se observa en la pantalla.

Si volvemos a usar otro printf, por ejemplo:

```
#include <stdio.h>
main()
{
printf( "Cadena" );
printf( "Segunda" );
} Obtendremos:
```

Cadena Segunda

Este ejemplo nos muestra cómo funciona printf. Para escribir en la pantalla se usa un cursor que no vemos. Cuando escribimos algo el cursor va al final del texto. Cuando el texto llega al final de la fila, lo siguiente que pongamos irá a la fila siguiente. Si lo que queremos es sacar cada una en una línea deberemos usar "\n". Es el indicador de retorno de carro. Lo que hace es saltar el cursor de escritura a la línea siguiente:

```
#include <stdio.h>
main()
{
printf( "Cadena\n" );
printf( "Segunda" );
}
```

y tendremos:

Cadena Segunda

También podemos poner más de una cadena dentro del printf:

printf( "Primera cadena" "Segunda cadena" );

Lo que no podemos hacer es inluir cosas entre las cadenas:

printf( "Primera cadena" texto en medio "Segunda cadena" );

Esto no es válido. Cuando el compilador intenta interpretar esta sentencia se encuentra "Primera cadena" y luego texto en medio, no sabe qué hacer con ello y da un error.

Pero, ¿qué pasa si queremos imprimir el símbolo " en pantalla? Por ejemplo imaginemos que queremos escribir: Esto es "extraño". Si para ello hacemos:

printf( "Esto es "extraño"" ); obtendremos unos cuantos errores. El problema es que el símbolo " se usa para indicar al compilador el comienzo o el final de una cadena. Así que en realidad le estaríamos dando la cadena "Esto es", luego extraño y luego otra cadena vacía "". La función printf no admite esto y de nuevo tenemos errores. La solución es usar \". Veamos:

printf( "Esto es \"extraño\"" );

Esta vez funcionará. Como vemos, la contrabarra '\' sirve para indicarle al compilador que escriba caracteres que de otra forma no podríamos. Pero, ¿y si lo que queremos es usar '\' como un carácter normal e imprimir, por ejemplo, Hola\Adiós?. La respuesta es volver a usar '\':

printf( "Hola\\Adiós" );

Esta doble '\' indica a C que queremos es mostrar una contrabarra.

Como hemos visto, la sintaxis de printf acepta un string de formato (const char *format) y cualquier número de argumentos. Estos se aplican a cada uno de los especificadores de formato contenidos en format. Un especificador de formato toma la forma

%[flags][width][.prec][h|lL] type.

El tipo (type) puede ser:

d, i entero decimal con signo

o entero octal sin signo

u entero decimal sin signo

x entero hexadecimal sin signo (en minúsculas)

X entero hexadecimal sin signo (en mayúsculas)

f coma flotante en la forma: [-]dddd.dddd

e coma flotante en la forma: [-]d.dddd e[+/-]ddd

g coma flotante según el valor

E como e pero en mayúsculas

G como g pero en mayúsculas

c un carácter

s cadena de caracteres terminada en '\0'

% imprime el carácter %

p puntero

Los flags pueden ser los caracteres:

+ Siempre se imprime el signo, tanto + como -

- justifica a la izquierda el resultado, añadiendo espacios al final

blank si es positivo, imprime un espacio en lugar de un signo +

# especifica que el argumento se convertirá según una forma alternativa

En el campo width se especifica la anchura mínima del valor de salida:

n se imprimen al menos n caracteres.

0n se imprimen al menos n caracteres y, si la salida es menor, se anteponen ceros

* La lista de parámetros proporciona el valor de anchura

Hay tres modificadores de tamaño, para los tipos int y double:

h imprime un entero short

l imprime un entero long

L imprime un double long

Si se usa el flag # con una conversión de formato de printf, tiene el siguiente efecto sobre

El argumento:

C s d i u no tiene efecto

0 antepone 0 a un argumento no nulo

x o X antepone 0x (o 0X) al argumento

e E f el resultado siempre contiene un punto decimal

g G igual que e E. Los ceros sobrantes se eliminan

scanf: lectura del teclado

Algo muy usual en un programa es esperar que el usuario introduzca datos por el teclado.

Para ello contamos con varias posibilidades: Usar las funciones de la biblioteca estándar, crear nuestras propias interrupciones de teclado (MS-DOS) o usar funciones de alguna biblioteca diferente. Vamos a estudiar la primera, usando las funciones de la biblioteca estándar.

Para utilizar la función scanf debemos incluir la directiva:

#include <stdio.h> al principio del programa puesto que esta función se halla definida en el archivo de cabecera stdio.h de la siguiente manera:

```
int scanf ( const char *format [, direcciones, ...] );
```

Como se observa, el uso de scanf es muy similar al de printf con una diferencia fundamental: nos da la posibilidad de que el usuario introduzca datos en vez de mostrarlos. No nos permite mostrar texto en la pantalla; por eso, si queremos mostrar un mensaje previo, usamos un printf delante. Un ejemplo:

```
#include <stdio.h>
main()
{
int num;
printf( "Introduce un número: " );
scanf ( "%i", &num );
printf( "Has tecleado el número %i\n", num );
}
```

Obsérvese que en la sentencia scanf la variable num lleva delante el símbolo &. Éste sirve para indicar al compilador cuál es la dirección (o posición en la memoria) de la variable en la que se almacenará el valor introducido. Es imprescindible no olvidar este símbolo.

scanf no mueve el cursor de su posición actual, así que en el ejemplo queda ( _ indica dónde se halla el cursor):

Introduce un número _

Esto es porque en printf no hemos puesto al final el símbolo de salto de línea '\n'. Además, hemos dejado un espacio al final de Introduce un número: para que cuando tecleemos el número no se "pegue" al mensaje.

Veamos cómo funciona scanf. Fijémonos que hay una cadena entre comillas. Ésta es similar a la de printf; sirve para indicarle al compilador qué tipo de datos estamos introduciendo.
Como en este caso se trata de un entero, usamos %i. Después de la coma se halla la variable donde almacenamos el dato, en este caso num.
Podemos preguntar por más de una variable a la vez en un solo scanf, de modo que es preciso poner un %i por cada variable:

```
#include <stdio.h>
main()
{
int a, b, c;
printf( "Introduce tres números: " );
scanf ( "%i %i %i", &a, &b, &c );
printf( "Has tecleado los números %i %i %i\n", a, b, c );
}
```

De esta forma, cuando el usuario ejecuta el programa debe introducir los tres datos separados por un espacio.
También podemos pedir en un mismo scanf variables de distinto tipo:

```
#include <stdio.h>
main()
{
int a;
float b;
printf( "Introduce dos números: " );
scanf( "%i %f", &a, &b );
printf( "Has tecleado los números %i %f\n", a, b );
}
```

A cada modificador (%i, %f) le debe corresponder una variable de su mismo tipo. Es decir, al poner %i el compilador espera que su variable correspondiente sea de tipo int. Si ponemos %f espera una variable tipo float.

Veamos ahora el uso de scanf con cadenas (strings). La función scanf almacena en una porción temporal de memoria (un buffer) lo que vamos escribiendo. Cuando pulsamos ENTER (o Intro, o Return) lo analiza, comprueba si el formato es correcto y, por último, lo almacena en la variable que le indicamos. Veamos el ejemplo:

```
#include <stdio.h>
main()
{
char cadena[30];
printf( "Escribe una palabra: " );
scanf( "%s", cadena );
printf( "He guardado: \"%s\" \n", cadena );
}
```

Ejecutamos el programa e introducimos la palabra "hola". Esto es lo que tenemos:

```
Escribe una palabra: hola
He guardado: "hola"
```

Si ahora introducimos "hola amigos" esto es lo que tenemos:

```
Escribe una palabra: hola amigos
He guardado: "hola"
```

Sólo ha recogido la palabra "hola" y se ha olvidado de amigos. ¿Por qué? Porque scanf toma una palabra como cadena y usa los espacios para separar variables.

Es importante siempre asegurarse de que no vamos a almacenar en cadena más caracteres que los que caben. Para ello debemos limitar el número de los mismos que va a introducir scanf. Si, por ejemplo, queremos un máximo de 5 caracteres, usaremos %5s:

```
#include <stdio.h>
main()
{
char cadena[6];
printf( "Escribe una palabra: " );
scanf( "%5s", cadena );
printf( "He guardado: \"%s\" \n", cadena );
```

```
}
```

Si introducimos una palabra de 5 caracteres (no se cuenta '\0') o menos, la recoge sin problemas y la guarda en cadena:

Escribe una palabra: Frodo

He guardado: "Frodo"

Si introducimos más de 5 caracteres cortará la palabra y dejará sólo 5.

Escribe una palabra: Gandalf

He guardado: "Ganda"

scanf permite controlar qué caracteres introducimos. Supongamos que sólo queremos recoger las letras mayúsculas:

```
#include <stdio.h>
main()
{
char cadena[30];
printf( "Escribe una palabra: " );
scanf( "%[A-Z]s", cadena );
printf( "He guardado: \"%s\" \n", cadena );
}
```

Guarda las letras mayúsculas en la variable hasta que encuentra una minúscula:

Escribe una palabra: Aragorn

He guardado: "A"

Escribe una palabra: GOLLUM

He guardado: "GOLLUM"

Veamos qué sucede cuando scanf no recoge todas las pulsaciones efectuadas. Tomemos el código:

```
#include <stdio.h>
main()
{
char c;
char nombre[20], apellido[20];
```

```
printf( "Escribe tu nombre: " );
scanf( "%[A-Z]s", nombre );
printf( "Lo que recogemos de scanf es: %s\n", nombre );
printf( "Lo que había quedado en el buffer: " );
while( (c = getchar())!= '\n' )
putchar( c );
}
```

Imaginemos que introducimos un nombre con mayúsculas y minúsculas:

Escribe tu nombre: GANdalf

Lo que recogemos de scanf es: GAN

Lo que había quedado en el buffer: dalf

Veamos otro ejemplo en el que hay que obrar con precaución respecto a scanf:

```
#include <stdio.h>
main()
{
char nombre[20], apellido[20];
printf( "Escribe tu nombre: " );
scanf( "%s", nombre );
printf( "Escribe tu apellido: " );
gets( apellido );
}
```

Cuando se teclea un nombre se obtiene:

Escribe tu nombre: Gandalf

Escribe tu apellido:

No hay opción a introducir el apellido puesto que en el buffer queda un ENTER (el de finalizar la introducción del nombre) que es recogido directamente por gets.

### 4.6.4.- Manejo de archivos.

Para comenzar a usar un archivo dentro de nuestro programa debemos asignarlo a una variable, en el siguiente listado podemos ver como se hace.

```
#include <stdio.h>
int main(){
 /*
 * Para usar un archivo tenemos que asociar una variable puntero
 * Al archivo para poder escribir o leer de el
 */
 FILE *manejador_archivo = NULL;
 /*
 * Ahora que tenemos la variable para asignarla al archivo procedemos a
 * abrir el archivo, el primer parametro es el nombre del archivo
 * con su localizacion exacta en el sistema de archivos.
 * El segundo parametro es la forma en la que se va a abrir el archivo,
 * En este caso estamos abriendo el archivo en modo de solo lectura
 */
 manejador_archivo = fopen("/home/nomar/archivo", "r");

 /*
 * Ahora debemos comprobar que el proceso de apertura fue exitoso
 * Para saber si nuestra apertura de archivo fue exitosa hacemos la
 * Siguiente comparacion, si el contenido de la variable es igual a NULL
 * El archivo no fue abierto.
 */
 if(manejador_archivo == NULL ) {
 printf("No fue posible abrir el archivo\n");
 return -1;
 }
 /*
```

```
 * Cuando ya no necesitemos trabajar mas con el archivo procedemos a cerrarlo
 */
 fclose(manejador_archivo);
 return 0;
}
```

En la línea 8 encuentra la declaración de la variable que usaremos para manipular el archivo, es una variable puntero de tipo FILE.

Haciendo uso de la instrucción fopen() le pedimos al sistema operativo que nos localice el archivo nombrado en el primer parámetro, y que lo abra en el modo solicitado en el segundo parámetro. Este archivo esta ubicado en un sistema de archivos *NIX, si el sistema de archivo fuese de Windows el nombre de archivo comenzaría con una unidad, por ejemplo "C:\\archivo".

Modos de apertura de un archivo

El segundo parámetro de fopen() indica la forma en que se manipulara el archivo. En el ejemplo anterior en la linea 16 estamos abriendo el archivo para solo leerlo. ¿Qué quiere decir? Que el archivo debe existir para poderlo abrir y que no se le añadirá o modificara el contenido actual del archivo.

Las formas de abrir el archivo son estas:

r — abre el archivo en modo de solo lectura.

w — abre el archivo para escritura (si no existe lo crea, si existe lo destruye).

a — abre el archivo para agregar información (si no existe lo crea).

r+ — abre el archivo para lectura/escritura (comienza al principio del archivo).

w+ — abre el archivo para lectura/escritura, sobre-escribe el archivo si este ya existe o lo crea si no).

a+ — abre el archivo para lectura/escritura (se sitúa al final del archivo).

Si vamos a trabajar con archivos binarios usamos la letra b así que los modos de acceso quedan "rb", "wb", "ab", "rb+", "wb+", "ab+".

Cerrar archivos

Luego de trabajar con un archivo lo recomendable es cerrarlo, esto hace que el sistema operativo escriba cualquier dato que pudiera estar en memoria a disco. Como se puede ver

en el ejemplo, en la linea 32 la instrucción fclose() cierra el archivo. Recibe un parámetro, que es, la variable con la que se maneja el archivo.

Rebobinar archivos

Cuando se trabaja con archivos se cuenta con lo que se llama un cursor, un número que indica la posición del archivo en la que nos encontramos. La instrucción rewind() regresa el cursor al principio del archivo. Recibe un parámetro, la variable con la que se maneja el marchivo.

El cursor de archivo indica el punto a partir del cual se leerán o escribirán datos.

Moverse en el archivo

Es posible mover el cursor de archivo haciendo uso de la instrucción fseek() que acepta 3 parámetros. El primer parámetro es la variable de archivo, el segundo parámetro es la posición en bytes, y el tercero es uno de estos tres:

SEEK_SET cuenta la posición a partir del principio del archivo.

SEEK_CURRENT cuenta la posición a partir de la posición actual del cursor.

SEEK_END cuenta la posición desde el final del archivo.

```
#include <stdio.h>
int main(){
 // Variable de archivo
 FILE *datos = NULL;
 // Abrimos el archivo datos = fopen("/home/nomar/datos", "a");
 // Comprobamos que de verdad abrio
 if(manejador_archivo == NULL ) {
 printf("No fue posible abrir el archivo\n");
 return -1;
 }
 // Rebobinamos el archivo y llevamos el cursor al principio.
 rewind(datos);
 // Ahora con fseek nos posicionamos donde querramos,
 // en este caso 300 bytes desde el inicio del archivo.
 fseek(datos, 300, SEEK_SET);
```

```
// Se ubica a 200 bytes del final del archivo
fseek(datos, 200, SEEK_END);

// Se mueve el cursor 3 bytes a partir de la posicion anterior
fseek(datos, 3, SEEK_CURRENT);

// Al final cerramos
fclose(manejador_archivo);

return 0;
}
```

¿Estoy al final del archivo?

¿Cómo saber si estamos al final de un archivo? La instrucción feof() nos retorna 1 o true si estamos efectivamente al final del archivo. Recibe un solo parámetro, la variable del archivo.

Archivos de texto

Los archivos de texto son archivos que están conformados por caracteres de texto plano (ASCII, UTF-8, o algún otro tipo de codificación de texto). Estos archivos podemos editarlos con un editor de textos sencillo como notepad en Windows o gedit en Linux.

Leer del archivo de texto

Para leer datos desde un archivo de texto contamos con las siguientes funciones:

- fgetc() permite leer un carácter desde el archivo, recibe un parámetro y es la variable del archivo. Devuelve el carácter leído.
- fgets() lee cadenas completas desde el archivo, hasta que encuentra un retorno de carro \n o un eof. Acepta tres parámetros el nombre de el arreglo de caracteres donde se va a guardar lo leído, cantidad de bytes a leer y variable de archivo.
- fscanf() funciona de la misma forma que un scanf() tradicional pero con un parámetro al principio que indica el archivo a leer.

```
/*
* Suponga que el contenido del archivo es:
```

```
* Gol
* Hola mundo
* Esto es un número con formato 3.141
* 3.141
*
* Con esto en mente, empecemos.
*/
#include <stdio.h>
int main(){
 // Variable archivo.
 FILE *entrada;
 // Variable numerica.
 float numero = 0.0;
 // Arreglo de caraceres para guardar cadenas.
 char buffer[100] = "";
 // Variable para guardar una letra.
 char letra;
 // Variable para los ciclos
 int i;

entrada = fopen("./archivo.txt", "r");

 // Si no podemos abrir el archivo, terminamos el programa.
 if(entrada == NULL) { printf("No se pudo abrir el archivo... \n"); return -1; }

 // Con este ciclo leemos la primera linea Gol con su respectivo
 // retorno de carro.
 for(i = 0; i<4;i++){ letra = fgetc(entrada); }
 // El resultado de usar fgets sera que en buffer quedara guardada
// la linea Hola mundo con su respectivo retorno de carro.
```

```
 fgets(buffer,100,entrada);

 // Ahora leemos Esto es un numero con formato 3.141
 fgets(buffer, 100, entrada);

 // Con fscanf leemos el numero (caracteres) y lo guardamos como una variable flotante.
 fscanf(entrada, "%f", &numero);

 fclose(entrada);

 return(0);
}
```

Escribir en el archivo de texto

☐ fputc() escribe un carácter en el archivo, recibe dos parámetros, el carácter a escribir y la variable de archivo.

☐ fputs() escribe una cadena en el archivo, recibe dos parámetros, la cadena a escribir, y la variable de archivo.

☐ fprintf() funciona de la misma forma que printf() pero su primer parámetro es lavariable de archivo.

```
#include <stdio.h>
int main(){
 // La variable de archivo
 FILE *archivito;
 // Variable para escribir datos en el archivo
 char buffer[100] = "Hola Mundo";

 // Variable numerica
 float numero = 3.1416;  archivito = fopen("./archivo.txt", "a+");

 // Si no podemos abrir el archivo, terminamos el programa
```

```
 If (archivito == NULL) { printf("No se pudo abrir el archivo... \n"); return -1; }
 fputc('G', archivito);
 fputc('o', archivito);
 fputc('l', archivito);
 fputc('\n', archivito);
 // La primera linea del archivo fue escrita y contiene la frase Gol\n
 // Intentemos escribir algo mas, una cadena!
 // Pero ojo, solo esta escribiendo Hola Mundo, falta el retorno de carro
// Para completar la linea. Fputs (buffer, archivito);
 // Ahora le toca el turno a fprintf
 fprintf ("\nEsto es un numero con formato %3.3f\n%3.3f", numero,numero);
 fclose(archivito);
 return 0;
}
```

Archivos binarios

Las anteriores funciones sirven para manejar archivos donde se guardan caracteres. Es decir si queremos guardar el numero 123456 ocupara 6 bytes, o 12 bytes dependiendo dela codificación que estemos usando. En ocasiones esto no es deseable. Queremos guardar la mayor cantidad de información ocupando la menor cantidad de espacio. Usando un formato binario nuestro número 123456 ocupara ahora 4 bytes (lo que mide un int en C).

Cuando trabajamos en formato binario muy a menudo trabajamos con estructuras, que nos permiten empaquetar un conjunto de datos y tratarlo como si fuese una unidad.

Para escribir en archivos binarios

fwrite() se usa para escribir datos binarios en un archivo, recibe tres parámetros, el primero es la variable a guardar, el segundo el tamaño de la variable a guardar, el tercero cuantas veces se va a guardar y por último la variable de archivo.

```
#include <stdio.h>
// Esta estructura se declara global para que todas las funciones del programa // tengan acceso a su definición.
struct formulario {
```

```
 char nombres[200];
 char apellidos[200];
 short edad;
};
int main(){
 // Declaramos una variable llamada persona, del tipo struct formulario
 struct formulario persona;

 // Variable de archivo
 FILE *salida;
 // Procedemos a rellenar la variable
 strcpy(persona.nombres,"Nomar Oscar");
 strcpy(persona.apellidos, "Mora Tovar");
 persona.edad = 40;
 // Abrimos el archivo notese la b para indicar que es binario salida = fopen("./datos.dat", "wb");

 // Si no podemos abrir el archivo, terminamos el programa.
 if(salida == NULL) { printf("No se pudo abrir el archivo... \n"); return -1; }
 // Escribimos 1 registro de tipo struct formulario
 fwrite(&persona, sizeof(struct formulario), 1, salida);

 return(0);
}
```

Para leer de un archivo binario

fread() se usa para leer datos binarios de un archivo, recibe tres parámetros, el primero es la variable donde se van a guardar los datos, el segundo el tamaño de la variable a leer, el tercero cuantas veces se va a leer y por ultimo la variable de archivo.

```
#include <stdio.h>
// Esta estructura se declara global para que todas las funciones del programa
```

```
// tengan acceso a su definición.
struct formulario {
 char nombres[200];
 char apellidos[200];
 short edad;
};
int main(){
 // Declaramos una variable llamada persona, del tipo struct formulario
 struct formulario persona;

 // Variable de archivo
 FILE *salida;

 // Abrimos el archivo notese la b para indicar que es binario salida = fopen("./datos.dat",
"rb");

 // Si no podemos abrir el archivo, terminamos el programa.
 if(salida == NULL) { printf("No se pudo abrir el archivo... \n"); return -1; }
 // Escribimos 1 registro de tipo struct formulario
 fread(&persona, sizeof(struct formulario), 1, salida);
 printf("El nombre es: %s %s", persona.nombres, persona.apellidos);
 printf("La edad es: %d", persona.edad);

 return(0);
}
```

# BIBLIOGRAFIA BASICA Y COMPLEMENTARIA


- Mike Banahan, Declan Brady Mark Doran. The C Book. Editorial Addison-Wesley; 2nd edición (1 Agosto 1991).

- Allen B. Downey. How to Think Like a Computer Scientist C++ Version. Editorial Createspace (20 marzo 2009).

- Bruce Eckel. Thinking in C++ Editorial: Pearson (10 febrero 1995).

- Frank B. Brokken. C++ Annotations 2012 Editor: University of Groningen 790 pág.