UNIVERSIDAD NACIONAL AUTONOMA DE NICARAGUA-LEON

Facultad de Ciencias y Tecnología

Departamento de Computación

TESIS PARA OPTAR AL TITULO DE

LICENCIATURA EN COMPUTACION

PLAN DOCENTE DE LA ASIGNATURA PROGRAMACION II DE LA UNIVERSIDAD URACCAN

DESARROLLADO POR:

Lola Lyssette Newball Crisanto

Juan Guillermo Pérez Munguía

Karla Patricia Pérez Pineda

TUTOR:

Msc. Raúl Hermógenes Ruiz Cabrera

León, 6 de Agosto del 2012

UNIVERSIDAD NACIONAL AUTONOMA DE NICARAGUA-LEON

Facultad de Ciencias y Tecnología

Departamento de Computación

PLAN DOCENTE DE LA ASIGNATURA PROGRAMACION II DE LA UNIVERSIDAD URACCAN

Desarrollado por:

Lola Lyssette Newball Crisanto

Juan Guillermo Pérez Munguía

Karla Patricia Pérez Pineda

TUTOR:

Msc. Raúl Hermógenes Ruiz Cabrera

León, 6 de Agosto del 2012

Indice

INTRODUCCION

En el año dos mil ocho se firmo un convenio de colaboración entre la Universidad Nacional Autónoma de Nicaragua de León (UNAN-León) y la Universidad de las Regiones Autónomas de las Costa Caribe Nicaragüense URACCAN y la Universidad Autónoma de Alcalá España con el objetivo de apoyar la carrera de Licenciatura en Informática Administrativa que oferta la universidad URACCAN-Bluefields. Este convenio está dirigido a impulsar la excelencia en la enseñanza que permitirá formar mejores profesionales en el área de la informática administrativa. Esta colaboración permitirá la especialización y actualización del personal docente. Por lo tanto este proyecto pretende ser una contribución para lograr los objetivos que se pretenden alcanzar con este convenio.

En el presente documento se desarrolla un plan docente para la asignatura de Programación II que será impartida en la carrera de Informática Administrativa que oferta la universidad URACCAN-Bluefields.

Este documento presenta la situación actual de esta asignatura, su relación con otras asignaturas, la metodología y material didáctico para impartirlas, la metodología de evaluación, así como la presentación del contenido teórico, el desarrollo y soluciones de las prácticas, y la bibliografía y referencias necesarias para la misma.

OBJETIVOS

Los objetivos del presente proyecto docente son:

- Desarrollar un plan docente viable que permita al estudiante Licenciatura en Informática Administrativa abarcar los conocimientos básicos de la programación en lenguaje C.

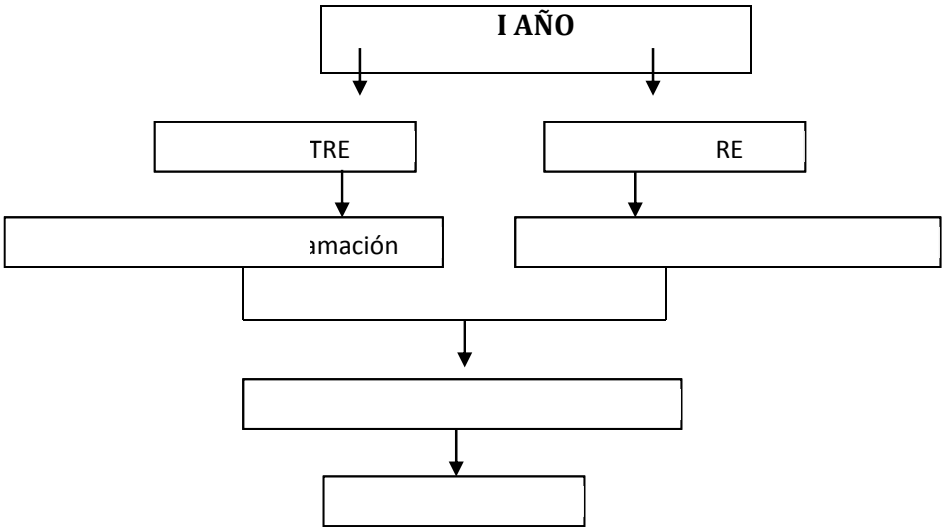
- Mejorar el contenido teórico y práctico de las asignaturas con respecto a los planes docentes ya existentes.

- Contar con un documento guía para impartir la asignatura de programación en lenguaje C

- Ofrecer al docente elementos para una mejor planificación y organización de la asignatura de Programación.

RELACION CON OTRAS ASIGNATURAS

El proceso de formación universitaria no consiste en la enseñanza y desarrollo de conocimiento, sino que es un proceso que consta de un conjuntos de conocimiento que están estrechamente relacionados, y que para una mejor organización, facilidad de compresión y aprendizaje se encuentra estructurados y organizados por asignatura, pero que al final cada estudiante debe adquirir para su futura aplicación en el campo profesional. Por tanto no se puede realizar un plan docente de una asignatura, ni mucho menos impartirla sin no se toma en cuenta la relación que existe entre las asignaturas, de forma tal que permita utilizar como base los conocimientos adquiridos previamente y adquirir nuevas bases para las asignaturas posteriores, y para la formación integral del alumno. Dicha Relación se muestra gráficamente a través de la figura siguiente y se detalla mas adelante.



Descripción del contenido de cada asignatura y su relación con otras asignaturas.

INTRODUCCION A LA PROGRAMACION

En este nivel el alumno ya tiene conocimiento de matemática, el cual le ayudara para su preparación y comprensión de esta asignatura.

Esta asignatura prepara al estudiante en el mundo de programación iniciando con programas sencillos.

Esta asignatura plantea los conocimientos sobre el ordenador, programas y lenguajes, así como el uso de algoritmo, diagrama de flujo, pseudocódigo para tener una mejor compresión para el desarrollo de programas.

A la vez que introduce al estudiante en la programación de alto nivel utilizando la programación en Turbo Pascal.

Así el estudiante aprenderá a elaborar programa en Turbo Pascal, compilar y correrlo en el ordenador. Conocerá su sintaxis, tipos de datos, funciones I/O.

PROGRAMACION I

Como el alumno en este nivel tiene una comprensión del ordenador y conoce los tipos datos fundamentales de la programación estructurada en Pascal y su sintaxis. Ahora se prepara al alumno para trabajar con funciones, procedimientos, sentencias de control, arreglos, registros. Para que pueda desarrollarse en el mundo de la resolución de problema mediante la programación.

PROGRAMACION II

Como el alumno ya tiene amplio conocimiento de programación en Pascal. Ahora le tocara aprender el lenguaje de programación en C, el cual se le ara más fácil la resolución de problema en este. Con este nuevo lenguaje aprenderá a trabajar con los distintos tipos de datos, funciones de entrada y salida, ficheros de cabeceras, sentencias de control, arreglos, estructuras de datos y punteros.

PROGRAMACION EN C

INTRODUCCION

La asignatura Programación II que se imparte en el Primer semestre del segundo año de la carrera de Licenciatura en Informática Administrativa con una duración de 64 horas semestrales.

El estudiante inicia la asignatura de programación II con conocimiento en programación de Turbo Pascal, la cual se ve en introducción a la programación y programación I, por lo que se le hace más fácil el aprendizaje del lenguaje de programación en C que imparte en la asignatura de programación II.

La asignatura de programación II es una de las mas importante para el estudiante de Licenciatura en Informática Administrativa pues es le dará la base necesarias al estudiante para que desarrolle las destrezas en su preparación para las asignatura superiores, pretendiéndose crear un habito de trabajo y la vinculación con el avance informático.

En los primeros se estudia las fases de desarrollo de un programa en lenguaje C, partiendo de una breve historia de este lenguaje, así como trabaja un programa, y las partes que componen el mismo, además las sintaxis de sus componentes.

En los siguientes temas se profundiza el estudio de las estructuras de un programa aquí el estudiante se tiene que familiarizar con todos los elementos de C, se le dará a conocer desde los ficheros de cabeceras, el trabajo con algunas funciones especiales, los distintos tipos de almacenamiento, las funciones de salida y entrada estándar, uso de formato, otras funciones que manipulan caracteres y funciones que manipulan el sistema operativo.

OBJETIVOS

Los objetivos de esta asignatura son:

- Explicar los conceptos básicos de programación estructurada.
- Sentar las bases necesarias para que se desarrolle las habilidades para programar.
- Crear conocimiento sólidos para sustentar el mejor aprendizaje de las asignaturas futuras de la carrera.

MATERIAL Y METODO DIDACTICO

Para impartir esta asignatura se utilizara la siguiente metodología:

Se impartirá una clase teórica semanal con una duración de dos horas, una vez a la semana, el cual se abordara de forma planificada el contenido teórico a desarrollar en el semestre.

Realización de trabajo investigativos que le permitan al alumno profundizar en la materia y a su vez desarrollar su capacidad autodidactica, búsqueda de información y uso de la información.

Las prácticas de laboratorio permitirán al alumno afianzar los aspectos teóricos vinculados con la práctica.

El material didáctico para impartir la clase será la pizarra y el datashow.

Se hará uso de la plataforma virtual de la universidad en el cual estarán disponibles todos los apuntes y material de apoyo sobre la misma. Así como la Bibliografía básica.

PLANIFICACION TEMPORAL

Para realizar una planificación objetiva del contenido de la asignación a lo largo del semestre se debe analizar la duración del periodo lectivo y obtener el número de horas que se dispone.

Partiendo del calendario académico propuesto por la URACCAN-Recinto Bluefields, esta asignatura se imparte en primer semestre del segundo año lectivo de la carrera de Licenciatura en Informática Administrativa, la cual cuenta con un total de 16 semanas, considerando los días feriados y otros incidencias que pueden ocurrir, podemos decir que se tiene 15 semanas netas para cumplir con el contenido de la asignatura tanto teórico como practico.

Para el desarrollo de la parte teórica de la asignatura se cuenta con un periodo de dos horas semanales, por lo que se tiene un total de 32 horas. Y para la práctica de laboratorio se cuenta con igual números de horas. Esto con el fin de ir afianzando los conocimientos teórico-prácticos de la asignatura de programación II en Lenguaje C.

La Planificación Temporal, como resultado de los elementos planteados anteriormente se expone en el cuadro que se presenta a continuación.

No	Tema	Teórico	Practico
1	Presentación	1	
2	Conceptos Básicos	3	2
3	Operadores y Expresiones	5	4
4	Sentencias de control	5	6
5	Funciones	4	4
6	Arreglos o Tipos Estructurados de datos	6	8
7	Punteros	8	8
	Total Horas	32	32

METODOLOGIA DE EVALUACION

La asignatura esta compuesta por las clases teóricas y prácticas que son realizadas en el laboratorio de la misma. Durante el semestre se realizan dos evaluaciones parciales que corresponderán al 100% de la nota. Estas evaluaciones están compuestas por una evalución teórica del 60% y el restante 40% a las prácticas de laboratorio.

Estas evaluaciones son de carácter obligatorio. Por lo que el alumno debe alcanzar un 60% para poder aprobarla.

En el siguiente cuadro podemos resumir lo anteriormente dicho:

Evaluación	Teórico	Practico	Total
I Parcial	60%	40%	100%
II Parcial	60%	40%	100%

Los exámenes escritos deberán contener preguntas teóricas directas que permitan evaluar el grado de compresión de los conceptos básicos que tiene el estudiante. Así como de preguntas analíticas, en el que alumno deberá de hacer un análisis a partir de los conocimientos teóricos adquiridos. Además se le plantearan preguntas de carácter prácticos.

El alumno deberá de realizar todas las prácticas establecidas en tiempo y forma para poder tener derecho al 40% correspondiente a laboratorio. El valor por práctica será proporcional entre el número de prácticas cumplidas.

CONTENIDO DEL TEMARIO

TEMA 0: PRESENTACION DE LA ASIGNATURA

- Objetivo de la asignatura
- Temario
- Material de estudio
- Apuntes y material de apoyo
- Bibliografías
- Tutorías
- Metodología de evaluación

TEMA 1: CONCEPTOS BASICOS

CONTENIDO

1. Reseña Histórica
2. Palabras Reservadas
3. Tipos de Datos
4. Constantes
5. Variables y Declaraciones
6. Expresiones y Sentencias
7. Elementos Generales de un Programa en C

TEMA 2: OPERADORES Y EXPRESIONES

CONTENIDO

1. Operadores Aritméticos
2. Operadores Monarios
3. Operadores Relacionales y Lógicos
4. Operadores de Asignación
5. Operadores de Condición
6. Prioridad de los operadores
7. Evaluaciones de Expresiones

TEMA 3: SENTENCIAS DE CONTROL

CONTENIDO

1. Sentencia if, Anidamiento de sentencia if, Estructura if.
2. Sentencia switch, sentencia break
3. Sentencia while, sentencia do, sentencia for bucles anidados.
4. Sentencia continúe, sentencia goto y etiquetas aplicaciones.

TEMA 4: FUNCIONES

CONTENIDO

1. Introducción
2. Defunción de una función
3. Llamada a una función, prototipo de una función.
4. Argumento por valor y por referencia

TEMA 5: ARREGLO O TIPOS ESTRUCTURADO DE DATOS

CONTENIDO

1. Introducción
2. Definición de arreglos, declaración
3. Cadenas de caracteres
4. Funciones para manipular cadenas de caracteres
5. Funciones para conversión de datos
6. Funciones para clasificación y conversión de caracteres
7. Matrices multidimensionales
8. Estructuras, array estructuras.
9. Uniones, campos de bits.

TEMA 6: PUNTEROS

CONTENIDO

1. Introducción
2. Concepto y creación de un puntero
3. Operadores * y &, operaciones con punteros
4. Operadores de asignación, operadores aritméticos.
5. Paso de un puntero a una función
6. Asignación dinámica de memoria
7. Punteros y formaciones unidimensionales.

TEMA I: CONCEPTOS BASICOS

OBJETIVOS

1. Conocer la historia y característica del lenguaje C
2. Conocer los elementos básicos para la construcción de un programa simple en C.
3. Conocer y trabajar con los tipos de datos, identificadores, expresiones, y evaluación de la misma.

CONTENIDO

- Reseña Histórica
- Palabras Reservadas
- Tipos de Datos
- Constantes
- Variables y Declaraciones
- Expresiones y Sentencias
- Elementos Generales de un Programa en C

BIBLIOGRAFIA

Caballos Sierra Fco. Javier C/C++, CURSO DE PROGRAMACION (Texto Base) 2 da. Edicion Editorial RA-MA España 2002 ISBN 84-480—6

Deitel – P.J Deitel. **COMO PROGRAMAR EN C/C++**. H.M. 2 da. Edicion Editorial Prentine Hall Hispanoamericana, S.A. Mexico-ENGLEWOOD CLUFFS-Londres SYDNEY

Herbert Schild C Manual de Referencia 4ta. Edicion Editorial McGRAW-HILL/Osborne McGraw-Hill

RESEÑA HISTÓRICA

El C es un lenguaje de alto nivel de propósito general. Sus principales características son:

- Programación Estructurada
- Abundante operadores y tipos de datos
- Eficiencia de ejecución del código generado (Entre los lenguaje bajo y alto nivel)
- Alta portabilidad (independencia del hardware subyacente)
- No impide tanta restricciones como los lenguaje de alto nivel, dando más libertad al programador
- Remplaza ventajosamente la programación en ensamblador
- Utilización natural de las funciones primitivas del sistema
- No está orientado a ningún área en especial.
- Producción de código objeto altamente optimizado
- Fácil de aprender.

El leguaje C fue inventado e implementado por primera vez por Dennis Ritchie en un DEC PDP-11 usando Unix, como sistema operativo.

C es el resultado de un proceso de desarrollo de los lenguaje BCPL desarrollado por Martin Richard se baso en otro lenguaje denominado B inventado por Kent Thompson 1970, tenía la intención de recodificar UNIX que en su fase de arranque estaba escrito en lenguaje ensamblador.

Dennis Ritchie en 1972 modifica B creando lenguaje C y rescribe UNIX en dicho lenguaje.

En 1983 se establece un comité para establecer ANSI (Instituto Internacional de Estándar Americano) el 14 de diciembre de 1989 se acordó estandarizarlo siendo adoptado también por ISO y desde entonces se denomino ANSI/ISO.

Para el año 1995 se realiza la enmienda 1, y el estándar de 1989 con la enmienda 1 se forma el documento base del estándar C++

Realización de un Programa en C

Pasos para crear un programa ejecutable en C:

1. Edición del programa o fichero fuente
2. Compilarlo o fichero objeto
3. Ejecutarlo
4. Depurarlo

La mayoría de los compiladores en C proporcionan un entorno de desarrollo que incluye un editor. Así también incluye compiladores independientes para los cuales es indispensable un editor.

Para esto debe estar almacenado en el disco duro las herramientas necesarias para que así pueda crearse un programa ejecutable, creándose los siguiente ficheros al escribir el siguiente programa saludo.c

Programa	Fichero creado
Editor	Saludo ó Saludo.cpp
Compilador C/C++	Saludo.obj en Windows o Saludo. o en Unix
Enlazador	Saludo.exe en Windows o a.out por omisión en Unix

En resumen se crearían los siguientes ficheros:

Fichero Fuente con extinción .c

Fichero Objeto con extensión .obj

Fichero ejecutable con extensión .exe

Ejemplo del programa saludo.c usando el editor de texto integrado con el compilador se guardara con el nombre y extensión .c o bien .cpp en C++

```
//Saludo
#include<stdio.h>
main()
{
printf("Bienvenido al Lenguaje de Programación en C");
}
```

Siendo su estructura la siguiente:

CODIGO FUENTE: programa que nosotros podemos escribir y guardar con la extensión .c o .cpp

CODIGO OBJETO: programa fuente pero traducido a lenguaje de máquina, se graba con la extensión .obj

PROGRAMA EJECUTABLE: es el programa objeto mas las ”librería de C” y se guarda con la extensión .exe y no necesita del programa que hemos utilizado para crearlo, para poderlo ejecutarlo.

En window se compila de con la orden: **cl**

cl saludo.c invocándose al compilar y enlazador de C

En Unix

cc saludo.c

Al ejecutarse proporciona por pantalla

Bienvenido al Lenguaje de Programación en C

Funciones de Biblioteca

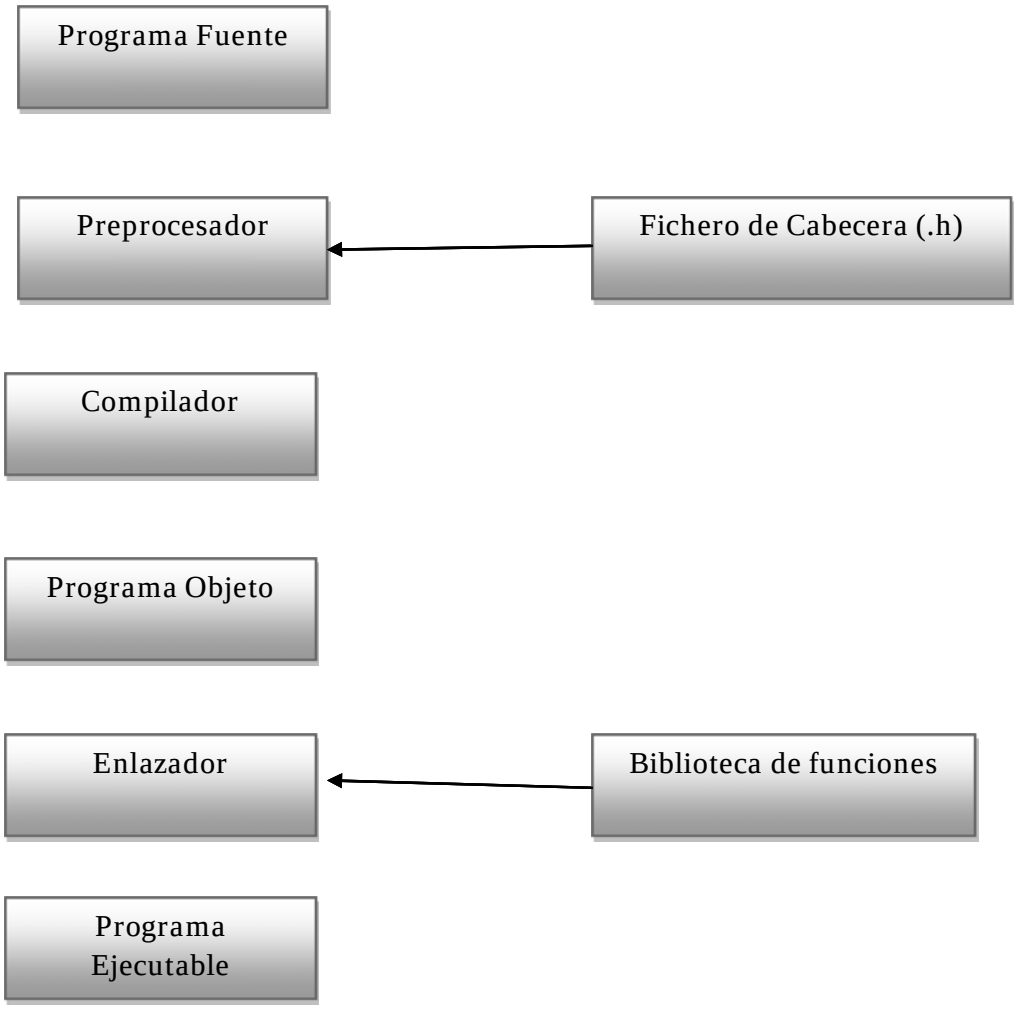
C carece de instrucciones de entrada y salida (E/S) para manejo de cadenas, de caracteres, etc. por lo que estas son incluida al compilarse el programa.

Una biblioteca es un fichero separado en disco (con extensión .lib o bien .a en Unix) el cual contiene las funciones a las que debemos hacer uso de ellas.

Ejemplo:

printf(“!Bienvenido a Lenguaje C!\n”) es la función de biblioteca para visualizar por pantalla la cadena que se coloca entre comillas.

A continuación mostramos los pasos que sigue el compilador para obtener un programa ejecutable a partir del fuente.



Mapa de Memoria

Un programa compilado en C crea y usa cuatro regiones de memoria lógicas diferentes, las cuales se presentan en el siguiente cuatro:

(Stack) Pila	(Región 4)
(Heap) Montón	(Región 3)
Variables globales	(Región 2)
Código de Programa	(Región 1)

Palabras Reservadas

El Lenguaje **ANSI C** está formado por 32 palabras claves o reservada que no pueden ser utilizada como nombres de variables ni de funciones que a continuación se detallan:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
const	**float**	**short**	**union**
continue	**for**	**signed**	**void**
default	**go to**	**sizeof**	**volatile**
do	**while**	**if**	**static**

Caracteres en C

El C cuenta con los elementos para trabajar como caracteres, secuencia de escape, tipos de datos operadores, etc.

Los caracteres en C pueden ser dígitos, letras, espacios en blancos, caracteres especiales, signos de puntuación y secuencias de escape.

LETRAS DIGITOS Y CARACTERES DE SUBRAYADO

Uso de los caracteres: para formar constantes identificadores y palabras claves. (C reconoce las letras mayúsculas y minúsculas en forma distintas)

Letras mayúsculas del alfabeto ingles

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Letras minúsculas del alfabeto ingles

a b c d e f g h i j k l m n o p q r s t u v w x y z

Dígitos

0 1 2 3 4 5 6 7 8 9

Subrayado

“ _”

Se denominan espacios en blanco todos los caracteres tabulares horizontales, verticales, avance de página y nueva línea.

Los espacios en blanco en exceso son ignorados por el compilador, estos se usan para hacer programas más legibles.

Ejemplo:

```
main()
{          Línea en Blanco                Espacio en Blanco

   printf ( “       Hola Dios te Ama \n” );
}
```

Otra secuencia de escape especial es el final del fin de fichero (End of File) en Windows Ctrl +Z y Ctrl +D en UNIX.

CARACTERES ESPECIALES Y SIGNOS DE PUNTUACION

Son caracteres que indican que un identificador es una función, matriz, o para identificar operación lógica o de relación

. , ; : ? ( ) [ ] { } <! / | + # % & * -= >

Secuencia de Escape

Existen una serie de caracteres que el editor lo toma como comandos, por lo que la manera de acceder es mediante una secuencia de escape la cual está formada por el carácter \ seguido de una letra o combinación de dígitos.

Secuencia	Nombre
\n	Nueva línea
\t	Tabulador horizontal
\v	Tabulador vertical
\b	Retroceso (backspace)
\r	Retorno de carro sin avance de línea
\f	Alimentación de pagina (solo impresoras) / nueva pagina
\a	Alerta pitido
\’	Comilla simple
\”	Comilla doble
\\	Barra invertida (backslash)
\ddd	Carácter ASCCI octal (0 al 7)
\xdd	Carácter ASCCI hexadecimal (0 al 9 o letra A a la Z)

Tipos de Datos

Existen siete tipos básicos de datos o propios del lenguaje.

Tipos enteros: char, int, short, long y enum.
Tipos reales: Float y double
Tipos especial void nada (realmente no es un tipo)

Tipos enteros: Los tipos **char** e **int** sirven para almacenar enteros y también valen para almacenar caracteres. Normalmente los números se almacenan en el tipo int y los caracteres en el tipo **char**, la realidad es que cualquier carácter puede ser representado como número (ese número indica el código en la tabla ASCII del carácter 'A' y 65 es lo mismo desde el punto de vista del C.

Tipos decimales

En C los números decimales se representan con los tipos **float** y **double.** La diferencia no solo está en que en el *double hay* números más altos, sino en la precisión.

Ambos tipos son de **coma flotante**. En este estilo de almacenar números decimales, la precisión es limitada. Cuántos más bits se destinen a la precisión, más preciso será el número. Por eso es más conveniente usar el tipo **double** aunque ocupe más memoria.

A los tipos de datos propios del lenguaje pueden tener distintos modificadores precediéndoles. Un modificador se usa para alterar el significado del tipo primitivo de forma que se ajuste más precisamente a las necesidades de cada momento.

Los modificadores son: **signed, unsigned**

La diferencia entre enteros con o sin signo está en cómo se interpreta el bit más significativo del entero. Si se especifica entero con signo el compilador genera código que asume que el bit más significativo será el bit de signo, si este es 0 entonces es positivo, si es 1 el entero es negativo.

Todos los tipos de datos definidos por el estándar de C.

tipo de datos	rango de valores posibles	tamaño aproximado en bits
char	127 a 255	8
Unsigned char	0 a 255	8
Signed char	-127 a +127	8
int	-32768 a +321767	16 0 32
Unsigned int	0 a65.535	16 0 32
Signed int	Igual que int	16 0 32
Short	-32768 a +32767	6
Long	-2147483648 a +2147483647	32
Unsigned long	0 a 4294967295	32
float	1E-37 a 1E+37 con 6 dígitos precisión	32
double	1E-37 a 1E+37 con 10 dígitos precisión	64
void	Sin valor	0

En la siguiente tabla se especifican unos ejemplos:

Tipo	Ejemplos de distintos tipos
char	char c=0; char = 'a'; char b=97; /* decimal*/ char b=0x61; /*hexadecimal*/ char b=0141; /*octal*/
Unsigned char	empleado para representación de caracteres ASCII
Signed char	igual que se usa sin calificar
int	int a=5000; int b= -40; int c= 0xf003
Unsigned int	no emplea el bit de signo int i=0, j=0; int a=5000

Signed int	emplea el bit de signo int b= -40
Short	short i=0, j=0;
Long	long a0 -1l; long b=125; long c=0x1f00230f
Unsigned long	long z=567899
float	float a=3.14592f; float b=2.2e +5f; float c=2/3f;
double	double a=3.14592; double b=2.2e +5; float c=2.07/3.0;
void	

Otro tipo de dato es el denominado **enum** que pertenece a los enteros, la declaración de un tipo enumerado es simplemente una lista de valores que pueden ser tomados por una variable de ese tipo. Los valores de un enumerado se representaran con identificadores, que serán las constantes del nuevo tipo.

Ejemplo:

```
enum dia_semana
{
      lunes,
      martes,
      miércoles,
      jueves,
      viernes,
       sábado,
       domingo
} hoy;
enum dia_semana ayer;
```

Las variables hoy y ayer son de tipo enumerado.

Creación de un enumerado

Crear un enumerado supone definir un nuevo tipo de dato, denominado tipo enumerado.

Sintaxis

```
 enum  tipo_enumerado
{
        /*  identificadores */
};
```

Cualquier identificador de la lista se le puede asignar un valor inicial entero por medio de una expresión constante. Los identificadores sucesivos tendrán valores correlativos a partir de este.

Reglas de los enumerados

- Dos o más miembros podrán tener el mismo valor
- Un identificador no podrá aparecer en más de un tipo
- No es posible leer o escribir directamente un valor de un tipo enumerado.

Ejemplo:

```
/*  enum.c  */
enum colores
{
azul, amarillo, rojo, blanco, negro
};
main()
{
     enum colores color;
     printf (“color”);  // leer color del teclado
     scanf (“%d”, &color);  // si se introduce 3
     printf ( “%d\n”, color);  // visualiza 3
}
```

¿para qué sirve un enumerado?

Estos tipos ayudan a acercar más al lenguaje de alto nivel a la forma de expresarnos.

Es mejor decir "si color es rojo" que "si color es 3"

Sinónimos de Tipos

Se utiliza la palabra typedef para declarar nuevos nombres de tipos de datos esto es sinónimos.

Ejemplo:

Typedef unsigned long ulong;

Tipo de dato Nuevo nombre

Literales

Un literal es una expresión de un valor de un tipo primitivo, o bien una cadena de caracteres ejemplo: 6,3.14, 'b', "hola", podemos decir que son valores constantes.

Literales Enteros

C permite un literal entro en base 10, 8,16, el signo + es opcional si el valor es positivo y signo (-) es obligatorio para los valores negativos.

Si literal es decimal y no tiene sufijo, su tipo es el primero de los tipos int, long int o unsigned long int.

Si es ocal o hexadecimal y no tienen sufijo, su tipo es el primero de los tipos int, unsigned int, long int, unsigned long int. También se puede indicar explícitamente el tipo de un literal entero con los sufijo L(long), U(unsigned) o UL(unsigned long).

Literales enteros, decimales digito del 0 al 9 pueden repetirse y el primero nunca cero.

Ejemplo.

4448 constante entera int

14247 L constante entera unsigned int

1000 L constante entera long

345 UL constante entera unsigned long

Los literales octales son uno o más dígitos del 0 al 7 teniendo como inicio siempre cero ejemplo:

0326 constante octal

Un literal hexadecimal son dígitos del 0 a 9 y literales A a la F (mayúsculas y minúsculas)) pueden repetirse siempre comienzan con 0x ó 0X.

Ejemplos:

256 decimal

0400 256 decimal a octal

0x 100 256 decimal a hexadecimal

-0400 -256 decimal a octal

-0x100 -256 decimal a hexadecimal

Literales Reales

Representan a los números reales, enteros y punto decimal.

Ejemplos:

-17.32

14.12

.008e3

27E-3

Se pueden añadir una f ó F ejemplo 17.32 F

Literales de un solo Carácter

Estos son de tipo char, se representan por un único carácter entre comillas simple.

‘ ‘ espacio en blanco

‘ x ‘ letra minúscula x

‘ \n ‘ cambio de líneas

‘\x07’ pitido

‘\ x1B ‘ carácter ASCII de Esc.

Literales de Cadena de Caracteres

Las cadenas de caracteres se encierran entre comillas dobles

Ejemplo " Bluefields Nicaragua"

"1.1416"

" " /* esto es cadena vacía */

Estas cadenas se almacenan en localizaciones sucesivas de memoria, cada carácter ocupa un byte y siempre finalizan con \0 (carácter nulo).

La cadena "león "representada gráficamente

l e o n \0

Identificadores

Son los nombres que damos a las variables y a las funciones de C. Lógicamente no pueden coincidir con las palabras reservadas. Además puesto que C distingue entre las mayúsculas y las minúsculas, hay que tener cuidado de usar siempre las minúsculas y mayúsculas de la misma forma (es decir, *nombre, Nombre* y *NOMBRE* son tres identificadores distintos).

El límite de tamaño de un identificador es de 32 caracteres (aunque algunos compiladores permiten más tamaño). Además hay que tener en cuenta que los identificadores deben de cumplir estas reglas:

- Deben comenzar por una letra o por el signo de subrayado (aunque comenzar por subrayado se suele reservar para identificadores de funciones especiales del sistema).
- Sólo se admiten letras (del abecedario inglés, no se admite ni la ñ ni la tilde ni la diéresis), números y el carácter de subrayado

Ejemplo:

Identificador correcto	Identificador incorrecto
Cont	3cont
Prueba2	Hola¡
Compra_dia	Compra…….dia

Los identificadores podrán Ser de cualquier longitud no todos los caracteres han de ser significativos.

Comentarios

Un comentario es un mensaje a quien lea el código fuente, el objetivo de esto es explicar las acciones del programa fuente.

Se trata de texto que es ignorado por el compilador al traducir el código. Esas líneas se utilizan para documentar el programa.

Esta labor de documentación es fundamental ya que sino al pasar un tiempo el código no se comprende bien. Todavía es más importante cuando el código va a ser tratado por otras personas, de otro modo una persona que modifique el código de otra lo tendría muy complicado. En C los comentarios se delimitan entre los símbolos /* y */

/ Esto es un comentario*

el compilador hará caso omiso de este texto/*

Como se observa en el ejemplo, el comentario puede ocupar más de una línea.

Existe otro tipo de comentario que sólo vale para comentarios de una línea. En ese caso los comentarios comienzan con los símbolos // y terminan con el final de la línea

// Este es c*omentario de una sola* línea.

Constantes

Una constante simbólica significa decirle al compilador de C el nombre de la constante y su valor, esto se hace antes de la función principal o main se utiliza la directriz # define

Sintaxis

```
#define  Nombre_constante  valor
Ejemplo
#define PI  3.1459
#define SL  ´ \n ´
#define MENSAJE “ pulse una tecla y continuar \n”
```

Const

Cuando se le antepone la palabra reservada **const** a una variable no puede ser modificada por el programa. (se puede iniciar con un valor).

El compilador es libre se situar esta variable en la memoria de solo lectura.

Const double pi=3.1416; /* esta variable después de iniciada no puede cambiar*/

¿Porque usar las constantes?

Se utilizan las constantes porque es más fácil de modificar los programas, una vez definida una variable en un programa si se usa esta muchas veces en el mismo, solo tenemos que modificar en donde la definimos y nos olvidamos donde y cuantas veces se ha utilizado en el programa.

Variablesy Declaraciones

Variables

Las variables sirven para identificar un determinado valor. En C es importante el hecho de tener en cuenta que una variable se almacena en la memoria interna del ordenador (normalmente en la memoria RAM) y por lo tanto ocupará una determinada posición en esa memoria.

Es decir si *saldo* es un identificador que se refiere a una variable numérica que en este instante vale 8; la realidad interna es que *saldo* realmente es una dirección a una posición de la memoria en la que ahora se encuentra el número 8.

Declaración de variables

En C hay que declarar las variables antes de poder usarlas. Al declarar lo que ocurre es que se reserva en memoria el espacio necesario para almacenar el contenido de la variable. No se puede utilizar una variable sin declarar. Para declarar una variable se usa esta sintaxis:

tipo identificador;

Por ejemplo:

int x;

Se declarara *x* como variable entera.

En C se puede declarar una variable en cualquier parte del código, basta con declararla antes de utilizarla por primera vez. Pero es muy buena práctica hacer la declaración al principio del código. Esto facilita la comprensión del código.

También es buena práctica poner un pequeño comentario a cada variable para indicar para qué sirve. Finalmente el nombre de la variable (el identificador) conviene que sea descriptivo. Nombres como *a, b* o *c*; no indican nada. Nombre como *saldo, gastos, nota,...* son mucho más significativos.

Nota: el nombre de la variable nada tiene que ver con su tipo.

Expresiones y Sentencias

SENTENCIAS

Los programas en C se basan en sentencias las cuales siempre se incluyen dentro de una función. En el caso de crear un programa ejecutable, esas sentencias están dentro de la función **main.**

Las normas a seguir son los siguientes:

1) Toda sentencia en C termina con el símbolo "punto y coma" (;)

2) Los bloques de sentencia empiezan y terminan delimitados con el símbolo de llave ({ y }). Así "**{**" significa inicio y "**}**" significa fin

3) En C hay distinción entre mayúsculas y minúsculas. No es lo mismo **main** que **MAIN**. Todas las palabras claves de C están en minúsculas. Los nombres que pongamos nosotros también conviene ponerles en minúsculas ya que el código es mucho más legible así.

EXPRESION

Una expresión es un conjunto de operandos unidos mediante operadores para especificar una operación, todas las expresiones cuando se evalúan retornan un valor.

Ejemplo:

a+b;

Cantidad * precio;

Z= x*x + 2*x*y + y*y;

Cuando se evalúan expresiones de operandos de diferentes tipos, C realiza conversiones solo para realizar las operaciones, los valores de los operandos al tipo de operando cuya precisión sea la más alta, ejemplo un int es más preciso que un char, un doble es más preciso que un int.

Elementos Generales de un Programa en C

Todo programa en C consta de una o más funciones, lógicamente una función no es más que un conjunto de instrucciones que realizan una tarea específica.

Si un programa en C está compuesto de varias funciones ¿Cuál es la función que da comienzo al programa? La función main o función principal que indica que cualquier programa en C contiene una función nombrada main.

Cada función debe contener:

1. **Directivas de pre-procesador**: Son las instrucciones que se le dan al compilador antes de compilar. Las dos directivas más usuales son #incluido y # define. Todas las directivas deben comenzar con símbolo ‘ # ‘, que indica al compilador que incluya las directivas antes de compilar la parte principal del programa.

    - # **include:** Indica al compilador que lea el archivo fuente que viene a continuación de ella y su contenido lo inserte en la posición donde se encuentra dicha directiva. Estas funciones se encuentran disponibles en las bibliotecas o archivos de cabecera que tienen asociado un fichero con extensión .h.

    - # **define:** Permite definir constantes simbólicas y macros.

2. **Declaraciones Globales**:
    - **Prototipos de Funciones**
    - **Declaraciones de Variables**

    Indican al compilador que las funciones definidas por el usuario o variables así declaradas son comunes a todas las funciones del programa. Estas deben situarse antes de la función main( ).

3. Función Principal **main( )**

    Esta es la función principal de nuestro programa, **NUNCA** debe faltar, ya que en ella van contenidas todas las instrucciones de nuestro programa.

Los comentarios: Son líneas que ignora el compilador pero sirven para documentar los programas. Pueden empezar con **/ * y** terminar con *** /,** estos pueden abarcar varias líneas.

Sintaxis de un programa en C

#include<*nombre_biblioteca.h***>**

[*#define identificador valor*]

[Declaraciones variables Globales]

[int] **main ( )**

{

Declaraciones variables locales /*Comentarios */

Sentencia(s)

[return valor]

}

Ejemplo:

Include < stdio.h > **/***esta línea le indica al compilador que incluya
Información acerca de las funciones de
Entrada y salida de la biblioteca estándar < stdio.h>***/***

/ * Define la función main, indicando que está definida para no
Recibir argumentos pero que retornara un valor de tipo entero al S.O *** /**

main ( )
{ // ← Aquí comienza el programa

printf (" Bienvenido al Curso de Lenguaje C ");
// llama a la función printf para que muestre en Pantalla un mensaje

Printf (" ¿Estás listo ?"); **//** llama a la función Printf para que muestre otro mensaje

Return 0; **//** retorna valor cero, indicándole al S.O que el programa termino exitosamente

} // ← Aquí termina el programa

TEMA II: OPERADORES Y EXPRESIONES

OBJETIVO

- Conocer con detalle los operadores en C y como están organizados.

CONTENIDO

1. Operadores Aritméticos
2. Operadores Monarios
3. Operadores Relacionales y lógicos
4. Operadores de Asignación
5. Operadores de Condición
6. Prioridad de los Operadores
7. Evaluación de Expresiones

BIBLIOGRAFIA BASICA

Caballos Sierra Fco. Javier C/C++, CURSO DE PROGRAMACION (Texto Base) 2 da. Edicion Editorial RA-MA España 2002 ISBN 84-480—6

Deitel – P.J Deitel. **COMO PROGRAMAR EN C/C++**. H.M. 2 da. Edicion Editorial Prentine Hall Hispanoamericana, S.A. Mexico-ENGLEWOOD CLUFFS-Londres SYDNEY

Herbert Schild C Manual de Referencia 4ta. Edicion Editorial McGRAW-HILL/Osborne McGraw-Hill

Operadores

Se trata de unos de los puntos más fuertes de este lenguaje ya que permite especificar expresiones muy complejas, por lo que le da más significado a los operadores que la mayoría de lenguajes de programación.

Se clasifican en: Aritméticos, relacionales, lógicos, unitarios, a nivel de bit, de asignación, de condición y otros.

Operadores Aritméticos

Estos operadores los resumimos en la siguiente tabla

Operadores	Operación
+	Suma operandos pueden ser enteros o reales
-	Resta operandos pueden ser enteros o reales
*****	Multiplicación operando pueden ser enteros o reales
/	División operando enteros o reales si operando son entero el resultado es entero, resto de caso resultado real
%	Modulo o resto de una división entera. Los operandos tienen que ser enteros

Ejemplos

```
Int x, y;
X=5, y=2;
printf("%d", x/y);        /*mostrará 2*/
printf("%d", x/y);       /*mostrará 1*/
```

Operadores Monarios

Estos operadores se aplica a un solo operando y son ! - ~ ++ y

Operando	Operación
~	Complemento a uno (cambia 1 a 0 y = por 1) carácter ASCII 126
-	Cambia el signo al operando (se calcula completamente a dos que es complemento a 1 mas 1)

Ejemplo

```
Int a=2, b=0, c=0;
c=-a ;          //c = -2
c = -b          //c = -1
```

Operadores de Relación

Sirven para realizar comparaciones. El resultado es verdadero o falso (0 o 1).

Operador	Operación
<	Menor que
>	Mayor que
<=	Menor o igual
>=	Mayor o igual
!=	Distinto de (no igual a)
==	Igual a

Ejemplo

Los operadores tienen que ser de un tipo primitivo

```
Int x=10, y=0, r=0;

r= x==y;        //r=0 (falso)
r= x > y;       //r=1 (falso)
r= x>!=y;       //r=1 (falso)
```

Un operador de relación equivale a una pregunta relativa de cómo son los operadores entre sí.

Esto es x==y (x es igual a y) si repuesta es si (verdadero 1) si no (falso 0)

Operadores Lógicos

Permiten agrupar expresiones lógicas. Estas son todas aquellas expresiones que obtienen como resultado verdadero o falso (1 ó 0)

Operador	Operación
&&	AND da como resultado verdadero si ambas operadores son verdaderos de lo contrario es falso
||	OR resultado es falso si ambos operadores son falso de los contrario es verdadero (| carácter ASCII 124)
!	NOT contrario del valor de verdad del operando

Tabla de verdad de operadores lógicos

p	q	p&&q	p||	!p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Operadores a Nivel de Bit

Estos operadores permiten realizar con sus operandos las operaciones AND, OR, XOR y desplazamiento bit a bit los operadores tienen que ser enteros.

Operador	Operación
&	AND a nivel de bit
|	OR a nivel de bit
^	XOR a nivel de bit
<<	Desplazamiento a la izquierda rellenado con cerros por la derecha
>>	Desplazamiento por la derecha rellenado el bit de signo por la izquierda

Para &, ^, | y ~ NOT (complemento a Uno) están gobernado por la misma tabla de verdad que su equivalente lógico excepto que trabaja a nivel de bit.

Tabla de la O exclusiva (^)

p	Q	p^q
1	0	1
1	1	0
0	1	1
0	0	0

Si los operandos tienen igual valor de verdad es falso de lo contrario es verdadero.

Ejemplo

```
char a=5, b;
char mask=0x01;  /*mascara del bit menos significativo */

b=a & mask;          /* b=1 */
b= a << 2;           /* b= 20 */
b=~a;                /* b= -6 */
```

Los resultado con operadores de relación y lógicos producen valores verdaderos (1) o falso (0), asi los resultados a nivel de bit producen valores arbitrarios.

Operadores de desplazamiento de bit >> y << desplazan todos los bit de una variable a la derecha o a la izquierda según se especifica.

Ejemplo:

unsigned char x	X a medida que se ejecuta cada instrucción	Valor x
x=7	0000111	7
x=x<<7	0001110	14
x=<<3	0111000	112
x=<<2	1100000	192
x=x>>1	0110000	96
x=x>>2	0011000	24

Uno de los métodos mas sencillos es el de complementar cada byte usando complemento a uno para invertir cada bit del byte asi:

Byte original 0 0 1 0 1 1 0 0

1er complemento a uno 1 1 0 1 0 0 1 1

2do complemento a uno 0 0 1 0 1 1 0 0

iguales

Dos complemento a uno produce el mismo número, el primero codifica y el segundo descodifica.

Operadores de Asignación

Sirve para dar un valor a una variable después de realizada la asignación.

Operador	Operación
++	Incremento
--	Decremento
=	Asignación simple
***=**	Multiplicación mas asignación
/=	División mas asignación
%=	Modulo mas asignación
+=	Suma más asignación
-=	Resta mas asignación
<<=	Desplazamiento a la izquierda mas asignación

>>=	Desplazamiento a la derecha más asignación
&=	Operando AND sobre bit mas asignación
|=	Operando OR sobre bit mas asignación
^=	Operando XOR sobre bit mas asignación

Los operando tienen que ser del tipo primitivo

Ejemplos

```
 Int x=0, n=10, i=1;
 n++       //incrementa n en 1
 ++n       //incrementa n en 1
x = ++ n   //incrementa n en 1 y asigna resultado a x
i + = 2          //igual que la operación i = i +2
x * = n – 3      //es igual a la operación x =x *(n -3) y no x =x
+n  - 3
x >>= 1    //igual a n=n>>1 desplaza el contenido de n a un bit a
la derecha
```

Otro ejemplo

```
#include<stdio.h>
main()
{
   int x1=9, x2=9;    //declaración de variables
    int y1, y2;       //declaración de variables

    y2=++x2;          //x2 se incrementa a 10 y se le asigna este
valor a y2
     printf("%d \n",x1);   //El valor de x1 es 10

      printf("%d \n",y1);  //¡¡¡¡¡¡¡El valor de y1 es 9!!!!!!!
      printf("%d \n",y2);  //El valor de y2 es 10
}
```

Otro ejemplo

```
#include<stdio.h>
main()
{
     float x=0, a=20, b=10, d=4;
     x=(b – a);
     --b;
    x *= (b – d) *d / (b – d);
    b++;
     printf("x =%g b= % g \n", x,b);
}
```

Resultado x=90, b=10

Operador de Condición

C tiene un operador muy potente el cual puede usarse para sustituir ciertas instrucciones de la forma (if then else) este es (**?**) llamado operador ternario.

Sintaxis: exp1**?** exp2:exp3

Actúa de la forma siguiente, evalúa exp1 si es cierta evalúa exp2 y toma ese valor toda la expresión. Si exp1 es falsa evalúa exp3 y toma ese valor toda la expresión.

Ejemplo

```
#include<stdio.h>
main()
{
    int x=5, y=4, mayor;
   mayor=(x >y) ? x : y;
   printf (“El mayor de los números es: %d”, mayor);
}
```

Otros Operadores

Operador Sizeof

Es un operador de tiempo de compilación devuelve la longitud en bytes del operando, esté operando puede ser el tipo o el identificador de una variable declarada.

Ejemplo

```
#include<stdio.h>
main()
{
    int a=0, t=0;
    t=sizeof (a);
     printf(“Tamaño del entero a es:%d bytes\n”,t);
    printf(“Tamaño del entero es:%d bytes\n”,int);
}
```

Resultado: Tamaño del entero a es:4 bytes
Tamaño del entero es:4 bytes

Operador Coma

La como se utiliza para encadenar varias operaciones, las cuales se ejecutan de izquierda a derecha.

Ejemplo

```
y=(x=3, x++);
aux=v1, v1=v2, v2=aux
```

Operador Dirección de (&)

El operador & (dirección de) permite obtener la dirección de su operando.

Ejemplo

int a =7;

```
printf("La dirección de memoria de a=%d,y el dato es %d\n",&a,a);
```

Resultado: La dirección de memoria de a= 17456 y el dato es 7

Operador de indirección (*)

El operador (*) (indirección) accede a un valor indirectamente a través de una dirección, permitiendo obtener el contenido de la dirección que apunta.

Ejemplo

```
#include<stdio.h>
int main()
{
 int destino, origen=10;
 int *m;
 m=&origen;
 destino=*m;
   printf("%d",destino);
}
```

Resultado 10

Prioridad y Evaluación de los Operadores

Se debe tener mucho cuidado al usar los operadores pues a veces podemos tener resultado no esperado si no tenemos cuidado en cuanto a su orden de evolución

En C una expresión como la siguiente f = a + b * c / d tiene un valor determinado por que conocemos de cómo se ejecutan las operaciones. Si se quiere otro resultado tendríamos que usar paréntesis para forzar el orden de ejecución de las operaciones.

Prioridad	Operadores	Asociatividad
Mayor	( ) [ ] . ->	Izquierda a derecha
	~ ! ++ -- * & sizeof	Derecha a izquierda
	/ * %	Izquierda a derecha
	+ -	Izquierda a derecha
	<<>>>>>	Izquierda a derecha
	<<= >>=	Izquierda a derecha
	== !=	Izquierda a derecha
	&	Izquierda a derecha
	^	Izquierda a derecha

	|	Izquierda a derecha
	&&	Izquierda a derecha
	||	Izquierda a derecha
	? :	Derecha izquierda
	= *= /= %= -= <<= >>= &= != ^=	Derecha izquierda

Operadores Especiales

El operador **.** (punto) permite hacer referencia a un campo de registro. Este es una estructura de datos que permite agrupar diferentes tipos de datos.

El operador flecha "**->**" permite acceder a un campo de un registro de cuando es un puntero el que señala dicho registro.

Los corchetes **[ ]** permite acceder a un elemento de un arreglo (array) el cual es una estructura que agrupa datos del mismo elemento.

CONVERSION ENTRE TIPOS

Cuando en una expresión se mezclan constantes, variables, de distintos tipos, todos se convierten a un tipo único, esto es solo para realizar las operaciones solicitadas; esta conversión es al tipo del operando cuya precisión es más alta. Si es una asignación convierte el valor de la derecha al tipo de la variable de la izquierda.

Reglas que se utilizan en la conversión:

1. De long doublé el otro operando a long doublé
2. De doublé el otro operando a doublé
3. De float el otro operando float
4. Si es char o short con o sin signo pasa a int además el tipo int puede representar a todos los valores del tipo original, o a unsigned int.
5. unsigned long el otro operando a unsigned long
6. long el otro operando long
7. unsigned int otro operando unsigned int.

Ejemplo

```
char c;
int i;
float f;
double d;
resultado = ( c / i) + (f * d) – (f + i);
```

int double float

double

doublé

otro ejemplo conversión del tipo implícita y explicita

```
doublé d, e, f=2.33;
int i=6;

e = f * i;   //e es double de valor 13.98
printf("Resultado es %f",e);
d = (int) (f * i);  //d es un doublé de valor 13.00
printf("Resultado es %f",d);
d = (int) f * i; //f se ha convertido a un entero truncado sus decimales
                 //d es doublé de valor 13.00
printf("Resultado es %f",d);
```

TEMA III: SENTENCIA DE CONTROL

OBJETIVOS

- Uso de funciones estándar de entrada y salida de caracteres
- Conocer las distintas sentencias de control.
- Saber en qué tipo de problema es más aceptable aplicar una sentencia de control.

CONTENIDO

1. Funciones de Entrada y salida de caracteres

    1) Función printf
    2) Función scanf
    3) Función getchar
    4) Función putchar
    5) Función gets
    6) Función puts
2. Sentencia If, anidamiento de sentencia if, estructura if
3. Sentencia switch, sentencia break
4. Sentencia while, sentencia do while, sentencias for bucles anidados
5. Sentencia continue, sentencia goto y etiqueta aplicaciones.

BIBLIOGRAFIA BASICA

Caballos Sierra Fco. Javier C/C++, CURSO DE PROGRAMACION
(Texto Base) 2 da. Edicion Editorial RA-MA España 2002 ISBN 84-480—6

Deitel – P.J Deitel. **COMO PROGRAMAR EN C/C++**. H.M. 2 da. Edicion Editorial Prentine Hall Hispanoamericana, S.A. Mexico-ENGLEWOOD CLUFFS-Londres SYDNEY

Herbert Schild C Manual de Referencia 4ta. Edicion Editorial McGRAW-HILL/Osborne McGraw-Hill

C cuenta con funciones de entrada y salida de caracteres de la salida estándar. Las cuales se pueden ejecutar desde cualquier sitio de un programa con simplemente escribir el nombre.

Función printf

Se puede utilizar esta función para escribir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de carácter. Esta función se ocupa de transportar datos de la memoria de la PC al dispositivo de salida estándar.
Veamos su formato**:**
printf(cadena de control, arg1, arg2….argn)

En donde cadena de control hace referencia a una cadena de caracteres que contiene información sobre el formato de salida y *arg1, arg 2, ….., arg n* son argumentos que representan los datos de salida. Los argumentos pueden ser constantes, variables, arreglos o expresiones mas complicadas.
Cada cadena de control debe comenzar por un signo de porcentaje (%) seguido de un carácter de conversión que indica el tipo de dato correspondiente.

Carácter de Conversión	Significado
%d	El dato es visualizado como un ***entero decimal* con signo**
%i	El dato es visualizado como un ***entero decimal* con signo**
%c	El dato es visualizado como un ***carácter (letra)***.
%s	El dato es visualizado como una ***cadena de carácter (varias letras)***.
%f	El dato es visualizado como un **valor de coma flotante (real) sin exponente.**
%ld	El dato es visualizado como un **entero largo con signo.**
%u	El dato es visualizado como un **entero decimal sin signo**
%lf	El dato es visualizado como un **valor de coma flotante de doble posición**.
%h	El dato es visualizado como un **entero decimal corto**
%o	El dato es visualizado como un **entero octal**
%x	El dato es visualizado como un H**exadecimal**
%e	El dato es visualizado como un **valor de coma flotante (real**) con exponente
%%	Imprime Porcentaje

Ejemplo

```
#include <stdio.h>

main()
{
  int a = 12345;
  float b = 54.865F;

     printf("%d\n", a);       /* escribe 12345\n */
     printf("\n%10s\n%10s\n", "abc", "abcdef");
     printf("\n%-10s\n%-10s\n", "abc", "abcdef");
     printf("\n");            /* avanza a la siguiente línea */
    printf("%.2f\n", b);     /* escribe b con dos decimales */
}
```

Función scanf

Se puede utilizar esta función para introducir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de carácter. Esta función se ocupa de capturar los datos del dispositivo de entrada estándar y los almacena en la memoria de la computadora.

En términos generales, la función scanf se escribe:

scanf (cadena de control, arg1, arg2, ……arg n)

En donde *cadena de control* hace referencia a una cadena de caracteres que contiene información sobre el formato de los datos de entrada y ***arg1, arg 2****, …..,* ***arg n****.* Son argumentos que representan los datos que se están ingresando.

En la cadena de control se incluyen grupos de individuales de caracteres, con un grupo de caracteres de conversión por cada dato de entrada. Cada grupo debe comenzar por un signo de porcentaje (%) seguido de un carácter de conversión.

Carácter de Conversión	Significado
D	El dato es un ***entero decimal***
I	El dato un ***entero decimal***
U	El dato es un **entero decimal sin signo**
H	El dato es un **entero decimal corto**
O	El dato es un **entero octal**
X	El dato es un H**exadecimal**
F	El dato es un **valor de coma flotante (real)**
E	El dato es un **valor de coma flotante (real**)
G	El dato es un **valor de coma flotante (real**)
C	El dato es un ***carácter***.
S	El dato es una ***cadena de carácter***.

Ejemplo

```
#include <stdio.h>

main()
{
     int horas, minutos;
  scanf("%d %*s %d %*s", &horas, &minutos);
  printf("horas = %d, minutos = %d\n", horas, minutos);
}
```

Función getchar

Se requiere leer un carácter desde el teclado (stdin), la cual no requiere ningún argumento.

Sintaxis

```
#include <stdio.h>
Int getchar (void)
//compatible con ANSI, UNIX y Windows

Forma general:

Variable de carácter = getchar();
```

Esta función devuelve el carácter leído ó un EOF si detecta el final del fichero.

Ejemplo

```
//Leer un carácter
#include <stdio.h>
main()
{
char car;
printf("Introduzca un carácter ");
car=getchar();
}
```

Funcion putchar

Para visualizar un carácter por pantalla (stdout) salida estándar, C tiene una función denominada putchar.

Sintaxis:

```
#include <stdio.h>
Int putchar (int c);
//compatible con ANSI, UNIX y Windows
```

Esta función devuelve el carácter leído ó EOF si ocurre un error.

Forma general:

putchar(variable de carácter)

Ejemplo

```
//Este programa Lee un texto desde el teclado. Presione ctrl +z para finalizar
#include<stdio.h>
int main()
{
        char car;
        while((car=getchar())!=EOF)
                putchar(car);
};
```

Ejemplo

//Este programa captura una línea de texto y lo visualiza por pantalla

```
#include<stdio.h>
main()
{
Char texto[80];
   int cont, aux;
   //Leer una linea de texto
 for(cont=0;(texto[cont]=getchar())!='\n';++cont)
       //apuntar al contador de caracteres
      aux=cont;
       //Escribir la línea de texto
          for(cont=0;cont<=aux;++cont)
              putchar(texto[cont]);
}
```

Funcion gets

Otra forma de leer una cadena de caracteres de el teclado (stdin) es usando la función gets

```
Sintaxis

#include<stdio.h>
Char * gets (char *var);
//Compatibilidad ANSI, Unix y Windows
```

Esta función proporciona mas comodidad para leer cadenas de caracteres que la funcion getchar y scanf.

Funcion puts

Otra forma de escribir cadenas en la salida estándar (stdout) es usar la función puts.

```
Sintaxis

#include<stdio.h>
Char * puts (char *var);
//Compatibilidad ANSI, Unix y Windows
```

La funcion puts de la biblioteca de C escribe una cadena en la salida estándar, esta funcion sustituye el carácter nulo \0 de final de cadena por el carácter \n nueva línea, es por eso que después de escribir la cadena se avanza automáticamente a la siguiente línea. Esta función retorna un valor positivo si se ejecuta satisfactoriamente y el valor EOF en caso de contrario.

Ejemplo

```
//Leer una línea de texto
#include<stdio.h>
main()
{
 char texto[80];

printf("Introd. Una línea de texto\n");
 gets(texto);
 putchar('\n');
 puts(texto)
}
```

Sentenciasde Control

En principio las sentencias de un programa en C se ejecutan secuencialmente, una después de la otra, de principio a fin del programa. C dispone de varias sentencias para modificar este flujo de secuencias; estas son denominadas sentencias de control agrupadas en bifurcación, estas permiten elegir entre dos o mas opciones según ciertas condiciones, los cuales permiten ejecutar repetidamente un conjunto de instrucciones tantas veces como se desee, cambiando o actualizando ciertos valores.

Sentencias if simple

Esta sentencia de control permite ejecutar si o no una sentencia simple o compuesta según cumpla o no una determina condición.

Esta sentencia tiene la forma:

```
if (condición)

   sentencia;
```

como trabaja:

Se evalúa la condición si es verdadera (diferente de cero), se ejecuta la sentencia; si es falsa (igual a cero), se salta la sentencia y prosigue con la siguiente línea de código. La sentencia puede ser simple o compuesta (bloque{…..})

Ejemplo

```
//Leer un numero
main()
{
   int num;
      printf("Escriba un numero");
      scanf("%d",&num);
     if (num > 0 )
     printf("El numero es positivo");
}
```

Sentencias if ....else

Esta sentencia permite realizar una bifurcación, ejecutando una parte u otra del programa según cumpla o no una cierta condición.

Forma general de representación

```
if (condición)
      sentencia_1;
else
      setencia_2;
```

Como trabaja:

Se evalúa la condición. Si el resultado es verdad (≠0), se ejecuta sentencia_1 y se prosigue en la línea siguiente de la sentencia_2; si el resultado es falso (=0), se salta la sentencia_1 y pasa a ejecutar la sentencia_2 y prosigue en la línea siguiente. La sentencias puede ser simple o compuestas (bloque{….})

Ejemplo

```
//Leer un numero
main()
{
    int num;
  printf("Escriba un numero");
    scanf("%d",&num);
   if (num > 0 )
printf("El numero es positivo");
else
printf("El numero es Cero o Negativo");
}
```

Anidamiento de la Sentencia if

Sentencias if anidadas

Una sentencia if puede incluir otros if dentro de la parte de la sentencia, a estas sentencias se le conoce como sentencias anidadas (una dentro de otra).

Ejemplo:

```
if (a >=b)
  if (b ¡=0)
c=a/b;
```

Aquí la interpretación no muy clara como con if… else, si (a >=b) es verdadero se ejecuta el siguiente if y si este es verdadero se ejecuta c=a/b. de lo contrario no se ejecuta nada y pasa a la siguiente línea del programa. Notar que los espacios en blanco y las tabulaciones son importantes, por lo que **else** aunque no aparezca, pertenece al **if** mas cercano.

Si se quiere que el **else** sea del primer if es necesario usar llaves asi

```
if (a >= b)
{
  if (b != 0)
     c= a/b;
}
else
  c=0;
```

Sentencias if ... else múltiples

Esta sentencia permite realizar una ramificación múltiple, ejecutando una entre varias partes del programa según se cumpla una condición.

```
Forma General:
if (condicion_1)
      sentencia_1;
else if (condicion_2)
        sentencia_2;
if (condicion_3)
      sentencia_3;
else if (condicion_4)
        sentencia_4;
……….. .
[else
      Sentencia_n]
```

Como trabaja:

Se evalúa condición_1, si el resultado es verdadero, se ejecuta la sentencia_1, si es falso se salta la sentencia_1 y se evalúa la condicion_2. Si el resultado es verdadero se ejecuta sentencia_2, si es falsa se evalúa la sentencia_3 y así en forma sucesiva. y si ninguna de las condiciones es verdadera se ejecuta la sentencia_n que es la opción por defecto (puede ser sentencia vacía, o bien eliminarse junto con el else) las sentencias pueden ser simples o compuestas.

Ejemplo

```
If ((hora >= 0) && (hora < 12))
  printf("Buenos Dias");
else if ((hora >=12) && (hora < 18))
         printf("Buenas tardes");
else if ((hora >=18) && (hora < 24))
        printf("Buenas noches");
else
        printf("Hora no valida");
```

Sentencias SWITCH CASE

Permiten seleccionar entre varios caminos para llegar al final. En este caso se pueden elegir un camino o acción a ejecutar de entre varios posibles que se debe de evaluar, llamada *selector*:

Esta sentencia desarrolla una función muy similar a la de **if …else**, con múltiples ramificaciones, además con muchas diferencias.

```
Forma general:

Switch (expresión)
{
      case opcion_1:
            sentencia_1;

      case opcion_2:
            sentencia_2;
      case opcion_3:
            sentencia_3;
      Case opcion_n:
            sentencia_n;
      [default :
            Sentencia;]
}
```

Como trabaja:

Se evalúa la expresión y se ve el resultado este es un valor entero, si dicho valor coincide con el valor constante opción_1, se ejecuta sentencia_1 y así si el valor entero coincide con cualquier de las opción_2, opción_3,…opción_n, se ejecutan las sentencias respectivas sentencia_2, sentencia_3, …..sentencia_n. Si ninguna de las expresiones coincide con el valor, entonces se ejecuta la sentencia después de **default.** Si se desea ejecutar solo un case se tiene que utilizar una que transfiera el control fuera del bloque switch, se puede poner break (otros elementos usados son **return, exit()**).

Es posible ejecutar una misma sentencia con distintos valores de case, solo basta poner varios case expresión seguidos para la misma sentencia, lo que no puede ocurrir es que exista para diferentes case expresión iguales.

Ejemplo

```
int dd=0, mm=0, aa=0;
printf("Introducir mes(##) año (####);");
scanf("%d %d",&mm,&aa)
switch(mm)
{
      Case 1: case 3: case 5: case 7: case 8: case 10: case 12:
       dd=31
      break;

      case 4: case 6: case 9: case 11:
        dd=30;
        break;
      case 2:
      //año bisiesto
      if ((aa%a==0) && (aa%100!=0)||(aa%400==0))
      dd=29;
      else
      dd=28
      break;
      default:
      printf("\n El mes no es válido\n");
      break;
}
```

Nota: La sentencia switch generalmente se recomienda usar cuando hay más de dos expresiones basados en una variable única de tipo numérico o carácter.

Bucles

En C existen también varias sentencias que permiten una serie de veces la ejecución de unas líneas de código, las cuales se realizan por un número determinado de veces hasta que se cumpla una condición de tipo lógico o aritmético a las cuales se le denomina bucles (son while, for, do while). Ya que nos permiten realizar una tareas un número determinado de veces.

Sentencia while

Esta sentencia permite ejecutar repetidamente, mientras se cumpla una determinada condición

```
Forma general:

while (condición)
      sentencia;
```

Como trabaja:

Mientras la condición sea verdadero la sentencia se repitiera y así sucesivamente. De lo contrario si la condición es falsa se salta la sentencia y se prosigue a la ejecución de la siguiente línea del programa.

Ejemplo:

```
#include<stdio.h>
main()
{
      int num;
      printf("\n Teclee un numero (0 para salir)");
      scanf("%d",&num);
      while (num !=0)
{
         if ( num > 0)      printf("El numero es positivo\n");
           eles       printf("El numero es negativo\n");
      printf("\n Teclee un numero (0 para salir)");
      scanf("%d",&num);
      }
}//Fin del programa
```

Sentencia do while

Esta sentencia funciona de forma análoga a la sentencia **while**, con la diferencia de que la evaluación de la condición se realiza al final del bucle, después haber ejecutado al menos una vez las sentencias entre llaves; estas se volverán a ejecutar mientras condición sea verdadera.

```
Forma general:

do
{
sentencia;
}while (condicion);
```

Sentencia puede ser una o un bloque de ellas se terminan la línea con (;) al final de la línea después del paréntesis que contiene la condición finaliza con (;).

Como trabaja:

1. Se ejecuta el bloque o sentencia simple.
2. Se evalúa la condición si resultado es falso se sale del bloque do y pasa ejecutar la siguiente línea del programa.
3. Si es verdadera se sigue ejecutando desde el paso 1.

Ejemplo

```
  int digito=0;
do
{
      printf("%d",digito);
      ++digito;
}while (digito <=9);
```

Sentencia for

La sentencia for es la usada y versátil. Además permite ejecutar una sentencia simple o compuesta un determinado número de veces conocido.

Forma general:

```
for (inicialización; condición; incremento)
      Sentencia;
```

La sentencia se ejecuta mientras la condición sea verdadera. Además antes de entrar en el ciclo se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento.
Es decir:

1. Se ejecuta la instrucción inicialización ejemplo. i=0
2. Se comprueba la condición
3. Si la condición es cierta, entonces se ejecuta la sentencia. Si la condición es falsa abandona el ciclo.
4. Tras ejecutar las sentencias, se ejecuta la instrucción incremento y se vuelve a al paso dos.

Ejemplo

```
for (i=0; i < 5; ++ i)
{
        printf("%i",i);
        if (i==3)
}
```

```
//Imprimir los múltiplos de 7 que hay entre 7 y 84

int k;
 for (k=7 ; k<=84 ;k+=7)
printf("%d",k);

//Imprimir los valores del 1 al 10

int i;
for (i=1; i<=10; ++i)
printf("%d",i);
```

Sentencia break

La sentencia break hace que el flujo del programa abandone el bloque en el se encuentra, también se puede utilizar para abandonar ciclos (while, for, do).

```
for (i=0; i < 5; ++ i)
{
        printf("%i",i);
        if (i==3) break;
}
```

Sentencia continue

La sentencia **continue** obliga que el programa comience el siguiente ciclo del bucle donde se halla, aunque no haya llegado al final de la sentencia del bloque.

Ejemplo

```
for (i=0; i <20; ++ i)
{

        if (i %3==0) continue;

        printf("%i",i);
}
```

Sentencia goto etiqueta

La sentencia **goto** etiqueta hace salir al programa a la sentencia donde se haya escrito la **etiqueta** correspondiente.

Ejemplo

```
main()
{
      int i, j;

      for (i=0; i <=5; i++)
        for (j=0; j<=10 ; j+=2)
         {
           if ((i==1) && (j >=7)) goto salida;
           printf("i vale %d y j vale %d\n",i,j);
         }
      salida:
      printf("Fin del programa\n");
}
```

Notar que etiqueta **salida** temina con dos puntos(:). La sentencia **goto** no es una sentencia de uso comun de los programadores de C, pues ello disminuye la claridad y legibilidad de los codigos fuentes. Esta sentencia fue introducida en C para mantener la compatibilidad con antiguos habitos de programacion, esta instrucción se puede sustituir por otra de mayor claridad.

TEMA IV: FUNCIONES

OBJETIVOS

- Definir el concepto de programación estructurada y modular
- Definir el concepto de funciones en C y describir sus partes
- Dar conocer las ventajas que se obtienen la utilización de función en un programa
  .

CONTENIDO

1. Introducción
2. Definición de una función
3. Acceso a una función
4. Prototipo de una función
5. Paso de argumento a una función

BIBLIOGRAFIA BASICA

Caballos Sierra Fco. Javier C/C++, CURSO DE PROGRAMACION (Texto Base) 2 da. Edicion Editorial RA-MA España 2002 ISBN 84-480—6

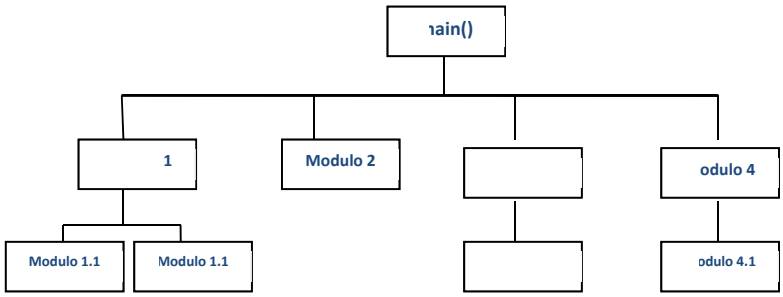
Deitel – P.J Deitel. **COMO PROGRAMAR EN C/C++**. H.M. 2 da. Edicion Editorial Prentine Hall Hispanoamericana, S.A. Mexico-ENGLEWOOD CLUFFS-Londres SYDNEY

Herbert Schild C Manual de Referencia 4ta. Edicion Editorial McGRAW-HILL/Osborne McGraw-Hill

Introducción

Podemos decir que la solución de un problema no se puede considerar en términos de sentencias directa de un lenguaje, si no como la naturaleza misma del problema, abriendo de alguna forma para dar lugar a la aplicación de funciones. Por lo que el uso de funciones permite dividir un programa grande en un cierto número de componentes más pequeños, cada una de las cuales con un propósito único e identificable

El diseño de programas, consiste en encontrar la solución de un problema mediante la aplicación sistemática de descomposición del problema en sub programas cada vez más simples aplicando el principio de divide y vencerá.



Definición de una Función

Es conjunto de sentencias que realizan tareas bien definidas y que se pueden llamar o ejecutar desde cualquier parte del programa.

Las funciones permiten al programador dividir un programa en partes bien definidas. Esto produce muchos beneficios tales como:

- Aislar mejor los problemas.
- Escribir programas correctos más rápido.
- Producir programas que son mas fáciles de mantener

Estructura de una función

```
[Tipo de retorno] [Nombre de la función] ([Parámetros formales])
{
Declaraciones variables locales;
 Sentencias;                          cuerpo de la función
 return Expresión
}
```

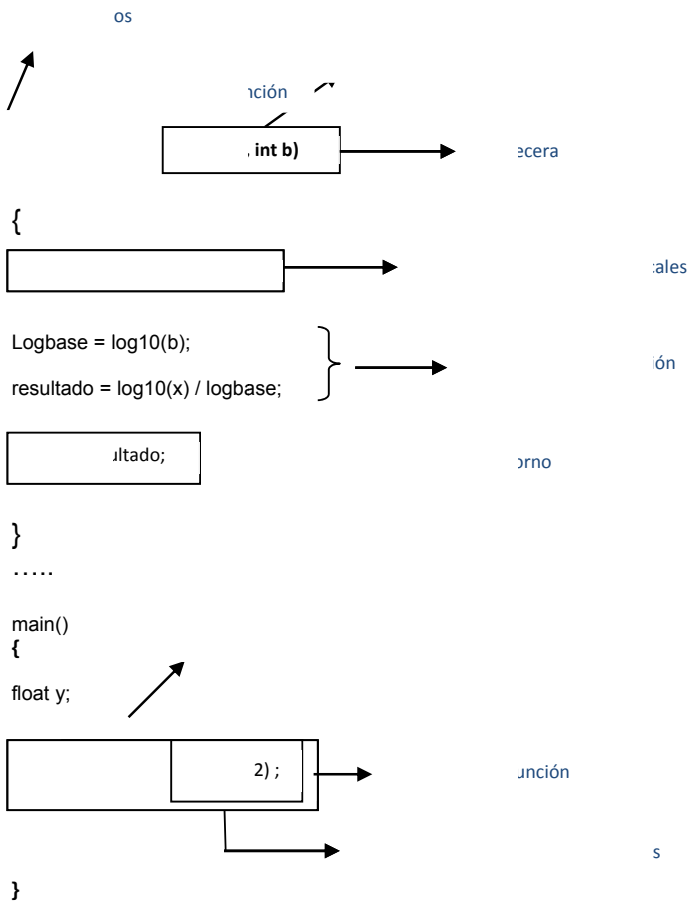
Descripción

Tipo de retorno: indica de qué tipo de datos es el valor devuelto por la función.
Nombre de la función: es el nombre que le hemos datos a la función, debe seguir las mismas reglas para la formación de variables.
Parámetros formales: Son variables que reciben los valores pasados en la llamada de la función. Estos permiten que se transfiera información desde el punto del programa en donde se llama a la función a esta. Y van separados por coma.

Declaración de variables locales: Aquí se ubican todas las variables y constantes que son declaradas para ser usadas solamente dentro de la función. Estas no son reconocidas fuera de la función, ya que son destruidas al terminar su ejecución.

return: son los tipos elementales de C, tales como int, char, float, o un puntero a cualquier tipo C, o un tipo struct, los cuales son devueltos por la función en el punto de llamada.

Ejemplo:



Llamada a una función (Acceso a una función)

La llamada una función es igual a ejecutarla. Cuando se llama a una función se hace desde otra función o de ella misma (caso de funciones especiales), además en el llamado se pasan los parámetros (parámetros actuales) o valores de la función, encerrados entre paréntesis y separados por comas.

Ejemplo

z = logaritmo (x , y);
Los argumento x, y son variables definidas y con su valor específico, **z** es una variable que tomara el valor después de ejecutada la función.

Función main()

Todo programa en C tiene una y solo una función principal denominada **main,** esta función es la que da pauta de entrada y salida del programa, y se define así:

```
Int main ( int argc, char *argv[] )
{
     //cuerpo de la funcion
}
```

main está diseñada para retornar un int por defecto, los dos parámetros son para guardar los datos cuando se ejecuta la función por línea de órdenes, es decir cuando desde el sistema operativo se invoque la función para ejecutarla.

Función prototipo (declaración de una función)

La declaración de una función o prototipo de la función permite conocer el nombre, el tipo de dato retornado, los tipos datos de los parámetros formales y opcionalmente sus nombres. No se define el cuerpo de la función. Una función no puede ser llamada si previamente no está definida o declarada. Además esta puede ser implícita o explícitamente.

Es implícita cuando la función es llamada y no existe prototipo de la función en este caso por omisión, se supone una función que retorna un int y no chequea el número de parámetros. Y explicita por que especifica el número y tipo de parámetros de la función, el valor de retornado.

Ejemplo

```
float fun_tres (int a1, int a2, float b1, float b2)
```

Donde los identificadores se encuentran separados por coma, esto puede también omitir a los identificadores como:

```
float captura(int, int, float, double);
```

Los identificadores de los parámetros de la prototipo y los de la definición de la función no necesariamente tiene que nombrarse de la misma forma. Ejemplo

```
float fun_tres( int a1, int a2, float x, double y);
float fun_tres( int dato1, int dato2, float sal, double z)
{
int r, z;
float w;
…..//
}
```

Además la lista de parámetros puede estar vacía así function_4( ) para ANSI C significa numero indeterminado de parámetros y en C++ que no hay parámetros, para la portabilidad se una mejor la palabra reservada **void.**

```
funtion_4(void)
```

Esta función no retorna ningún valor. Se lo pone **void** como valor devuelto `void funtion_4(void)`

Las funciones de la biblioteca de C no necesitan que se ponga el prototipo pues esta en los ficheros de cabecera .h si se utiliza una función de biblioteca como scanf o printf estas se incluyen en el fichero stdio.h en la sintaxis de las funciones nos indican en que fichero.h se encuentran declaradas.

Paso de Argumento a una función

Hay dos formas de pasar los parámetros actuales a sus correspondientes parámetros formales, cuando se efectúa la llamada a una función:

Por valor: Lo cual significa transferir una copia del valor del parámetro actual a su correspondiente parámetro formal. Esta operación se efectúa automáticamente cuando se llama a la función, y en el cual no se modifica el valor del parámetro actual.

Por referencia: Lo cual significa que lo transferido al parámetro formal no es el valor, sino la dirección de la variable (parámetro actual) que contiene dicho valor. Esta operación se efectúa automáticamente cuando se llama a la función, y en el cual si se modificar el valor del parámetro actual.

Para pasar un parámetro por referencia, se pasa la dirección del parámetro actual a su correspondiente parámetro formal, el cual tiene que ser un puntero. Para ello, hay que usar el operador unario de dirección **&** antes del nombre del parámetros actual y el operador de indireccion ***** antes del nombre del parámetro formal.

El operador de dirección **&,** permite obtener la dirección de una variable en la memoria RAM. El operador de indireccion *****, se usa para definir a un puntero, el cual es una variable que se usa para representar la posición de otra variable.

El método de paso por valor, permite transferir constantes, variables y expresiones; el método de paso por referencia, solamente se permite transferir las direcciones de variables de cualquier tipo, arreglos y funciones.

Ejemplos

/*Pasando parámetros por valor***/**

```
#include<stdio.h>
float media (float x, float y);
main()
{
 float num1, float num2, med;
 printf ("Introd. Dos numerous");
 scanf ("%f %f",&num1,&num2);
med=media(num1,num2);//llamado a la función con sus parámetros
                  //actuales
 printf("La media de dos números es:%f",med);
}


//definición de la función
float media (float x, float y)
{
 return ((x + y)/2);
}
```

/*Pasando parámetros por referencia***/**

```
#include<stdio.h>
void intercambio (int *, int *);
main()
{
 int a=20, b=30;
 intercambio(&a,&b); //Estos son pasados por referencia
 printf("A vale %d y b vale %d \n",a,b);
}
void intercambio (int *x, int *y)
{
int z=*x;
*x =*y;
*y = z;
}
```

Resultado: a=30 y b=30

Los cambios se han realizado porque se han puesto los parámetros como punteros a enteros estos reciben las direcciones respectivas &a, &b. Esto nos indica que se ha trabajado con las direcciones de las variables.

Accesibilidad de una Variable

Cada función puede definir sus propias variables locales definiéndolas en su cuerpo. C permite además, definir variables fuera del cuerpo de la función: variable globales.

Variables Locales
Estas son definidas dentro de un bloque ({….}) por lo que solo pertenecen a ese mismo bloque que puede ser una función o una sentencia compuesta. Estas puede ser tres tipos:

Automáticas: se crean al entrar en el bloque y se destruye al salir de él. Se le antepone la palabra **auto.** Por defecto todas las palabras locales en un bloque son automáticas.

Estáticas: No son destruidas al salir del bloque y su valor permanece inalterable cuando se vuelve a él. Se le antepone la palabra **static** al tipo de variable declarada.

Registro: Son variables estáticas de tipo char o int, que se almacenan en un registro de la CPU en vez de la memoria. Se le antepone la palabra **register**

Los bloques de códigos más normales son las funciones

Ejemplo

```
int suma (int x, int y)
{
int c;
c= x + y;
return c;
}
```

Variables Globales

A diferencias de las variables locales, estas se declaran fuera del cuerpo de cualquier función y son accesibles desde cualquier punto del programa posterior a su declaración manteniendo su valor a lo largo de la ejecución del programa.

Ejemplo

```
#include<stdio.h>
int cuenta;   //variable global
void fun1 (void);
void fun2 (void);
int main(void)
{
 cuenta=100;
```

```
 fun1();
 return 0;
}

void fun1(void)
{
int temp;
temp=cuenta;
 fun2()
 printf("Cuenta es %d",cuenta);
}

void fun2(void)
{
 int cuenta
 for(cuenta=1; cuenta>10;cuenta++)
 putchar('.');
}
```

Como se observa fun1 usa cuenta sin declararla internamente, mientras fun2 usa cuenta como variable propia que no tiene que ver con variable cuenta global.

Calificación de Variables Globales

Con las variables globales se puede utilizar los calificadores **static** (almacenamiento estático) y **extern** (almacenamiento externo), o bien omitir el calificador, en cuyo caso se supone extern.

Una variable global puede hacerse accesible antes de su definición o en otro fichero fuente, utilizando la calificación extern.

Ejemplo

```
//Fichero fuente uno.c
#include<stdio.h>
void func1();
void func2();
extern int var;
main()
{
 var++;
 prinft("%d \n",var); //se escribe 6
func1();
}

Int var=5; //definición de var;

Void func1()
{
Var ++;
printf("%d \n",var); //escribe 7
func2();
}
```

```
//Fichero fuentes dos.c
#include<stdio.h>
Extern int var; //se refiere a la variable externa que se
encuentra en uno.c
void func2()
{
 var++;
 printf("%d",var);//se escribe 8
}
```

En estos ejemplos existe tres declaraciones de var, dos en el fichero uno.c y uno en el fichero dos.c; las variables locales solo se inicializan una vez, como están declarada extern en el fichero uno.c por tanto se puede usar en main antes de definirse.

Calificación de las Funciones

Toda función tiene carácter global; puede definirse dentro de un bloque.

Las funciones declaradas **static** son accesibles en el fichero fuente en que se están definidas.

Si se declaran **extern** entonces pueden ser accesibles en todos los ficheros fuentes que componen el programa.

La declaración **static** o **extern** se puede hacer en la prototipo o bien en la definición de la función.

TEMA V: ARREGLOS

OBJETIVOS

- Definir el concepto y tratamiento de arreglos
- Crear y trabajar con matrices de diferente dimensión.
- Conocer funciones que manipulan las cadenas de caracteres.
- Saber definir los datos estructurados
- Trabajar con arreglos estructurados.

.

CONTENIDO

6. Introducción
7. Definición de arreglos
8. Procesamientos de arreglos
9. Paso de un arreglos a una funciones
10. Arreglos multidimensionales
11. Datos estructurados y uniones

BIBLIOGRAFIA BASICA

Caballos Sierra Fco. Javier C/C++, CURSO DE PROGRAMACION (Texto Base) 2 da. Edicion Editorial RA-MA España 2002 ISBN 84-480—6

Deitel – P.J Deitel. **COMO PROGRAMAR EN C/C++**. H.M. 2 da. Edicion Editorial Prentine Hall Hispanoamericana, S.A. Mexico-ENGLEWOOD CLUFFS-Londres SYDNEY

Herbert Schild C Manual de Referencia 4ta. Edicion Editorial McGRAW-HILL/Osborne McGraw-Hill

INTRODUCCIÓN

Una estructura de datos o tipo de dato estructurados es un tipo de dato construido a partir tipos de datos simples.

Un dato de tipo estructurado está compuesto por una serie de datos de tipos elementales que tienen alguna relación existente entre ellos. Una estructura de datos se dice que es **homogénea** cuando todos los datos elementales que la forman son del mismo tipo. En caso contrario se dice que la estructura es **heterogénea**.

Las estructuras de datos más empleados son: arreglos, registros (estructura), listas y árboles. El arreglo es la estructura de datos más usual. Existe en todos los lenguajes de programación.

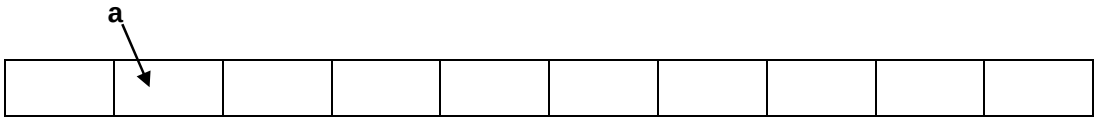
Definición de arreglos

Es un conjunto de elementos, todos los cuales son del mismo tipo. Estos elementos todos tendrán el mismo nombre, y ocuparan un espacio contiguo a la memoria. O es un conjunto finito ordenado de elementos homogéneos.

Ejemplo

Doublé a[10];

Se reserva un espacio para 10 variables de tipo doublé, las 10 variable se llaman **a** y se conoce a los datos por medio del nombre de la variable matriz y del subíndice sea este uno o varios encerrado entre corchetes.



Procesamiento de Arreglos

Declaración de un arreglo

Al igual que otras variables, antes de utilizarlas hay que definirlas, la definición de una arreglo de una dimensión (unidimensional) se realiza de la forma siguiente:

[Tipo de dato] [Nombre][Tamaño];

Tipo Dato: Indica el tipo de dato de los elementos del arreglo. Puede ser cualquier tipo excepto void;

Nombre: Es un identificador que nombra al arreglo.

Tamaño: Es una constante que especifica el numero de electos del arreglo. El tamaño puede omitirse cuando se inicializa el arreglo o cuando se declara como un parámetro formal en una función.

Los elementos se enumeran desde o hasta [Tamaño-1]

Acceder a los elementos de un arreglo

Para acceder a los elementos de un arreglo basta con poner el nombre del arreglo seguido del subíndice entre corchetes. En C no se puede operar con todo un arreglo como una única entidad, hay que tratar sus elementos uno a uno por medio bucles for, while; los arreglos (mejor dicho los elementos de los arreglos) se utilizan en las expresiones de C como cualquier otra variable.

Ejemplo:

a[5]=8;
a[9]=30*a[5];
a[0]=3*a[9]-a[5]/a[9];

Ejemplos:

```
#define elementos 10

int m[elementos];          //Crea un arreglo m
int i=0;
for(i=0; i<elementos; i++)
  {
  printf("m[%d]=",i);
 scanf("%d",&m[i]);
}
```

Inicialización de un arreglo

int vector[ ]={2,6,4,7,9};

Veamos el siguiente ejemplo de suma de vectores.

```
//El siguiente programa suma dos vectores
#include<stdio.h>
#define N 6
int main()
{
int i;  //declaramos i como índice de los vectores.
int v1[N],v2[N],v3[N];//declaramos tres vectores del mismo tamaño
//Lectura del vector 1
printf("Lectura del Primer Vector:");
for(i=0; i < N; i++)
 {
 printf("vector[%i]",i+1);
scanf("%d",&v1[i]);
 }
//Lectura del vector 2
```

```
printf("Lectura del Segundo Vector:");
for(i=0; i < N; i++)
 {
 printf("vector[%i]",i+1);
scanf("%d",&v2[i]);
 }

//Calculamos la suma de los dos vectores
for(i=0; i < N; i++)
 v3[i]=v[i]+v[2]; //suma de los vectores

printf("\n  Visualizamos  el  resultado  de  la  suma  de  los
vectores\n");
for(i=0; i < N; i++)
 printf("%i",v3[i]);
}//fin del main
```

Cadenas de caracteres

Una cadena de caracteres es un array de tipo char con algunas particularidades que conviene resaltar. Las cadenas suelen contener texto (nombre, frases, etc.), y este texto se almacena partiendo de la parte inicial de la cadena (a partir de la posición del arreglo). Para separar la parte que contiene texto con la utilizada, se utiliza el carácter fin de cadena ó carácter nulo ('\0') según el código ASCII este código se introduce automáticamente al leer o inicializar las cadenas de caracteres.

Ejemplo

```
char  ciudad[10]="Leon";
```

A los 4 caracteres de la cadena "Leon" se le integra el 5to que es \0 el resto de los elementos no se utilizan.

Las cadenas de caracteres se comportan como los arreglos numéricos. Estos tienen asociado un entero entre 0 y 255 (código ASCII). Por ejemplo 'L' le corresponde el valor 76.

Funciones que manipulan Cadenas

La biblioteca de C, contiene una gran cantidad de funciones que trabajan con cadenas de caracteres, desde funciones que leen, escriben, copian o bien convertir cadenas de caracteres.

Función C	Descripción
strcpy	Copia cadena
strcmp	Compara cadenas
strlen	Longitud o numero de caracteres
atof	Convertir una cadena a doublé
atoi	Convertir una cadena a int
atol	Convertir una cadena a long
sprintf	Convertir un valor desde cualquier formato admitido por printf a cadena de caracteres.
toascii	Convertir a ASCII
tolower	Convertir un carácter a minúscula
toupper	Convertir un carácter a mayúscula

Analizaremos cada una de ellas comenzando por unas funciones que leen y escriben cadenas de caracteres.

Para leer o visualizar el contenido de una cadena de carácter usamos las funciones scanf y printf ejemplo:

```
char nombre[41];
scanf(“ %s”,nombre);
printf(“%s\n”,nombre);
```

En este caso scanf no requiere ser precedido por & pues el nombre de un array es la dirección de comienzo del primer elemento del arreglo.

Si escribiese la siguiente cadena de carácter :Francisco Javier, cuando se visualiza su resultado solo aparece Francisco, para que se puede visualizar toda la cadena con los caracteres vacios se tendría que cambiar el formato en la función scanf así:

```
scanf(“%[^\n]”,nombre); //lee hasta encontrar un \n
```

Funciones strcpy, strcmp, strlen

strcpy

Sintaxis:

```
#include<string.h>
Char *strcpy (char *cadena1, const char *cadena2);
//Compatibilidad con ANSI, UNI X y Windows
```

Esta funcion copia el contenido de cadena2 en cadena1 incluyendo el carácter de fin de cadena y retorna un puntero a cadena1.

strcmp

Sintaxis:
```
#include<string.h>
char *strcmp (const char *cadena1, const char *cadena2);
//Compatibilidad con ANSI, UNI X y Windows
```

Esta función compara cadena1 con cadena2 de manera lexicográfica y devuelve un valor:

< 0 si la cadena1 es menor que cadena2
=0 si cadena1 es igual a cadena2
>0 si cadena1 es mayor que cadena2

Es decir esta función nos va permitir compara las cadenas para saber su orden alfabético, además diferencia las mayúsculas de las minúsculas, las mayúsculas están primero que las minúsculas por su valor en el código ASCII.

strlen

Sintaxis:

```
#include<string.h>
size_t strlen (char *cadena);
//Compatibilidad ANSI, UNIX y Windows.
```

La funcion strlen devuelve la longitud en bytes de cadena, no se incluye el carácter nulo (\n), **size_t** es tipo **unsigned int**

Funciones para conversión de datos

Clasificaremos algunas funciones para convertir cadenas a números y viceversas.

atof convierte una cadena de carácter a un valor double

Sintaxis:
```
#include<stdio.h>
double atof(const char *cadena);
//Compatible ANSI, UNIX, Windows
```

atoi convierte una cadena de carácter a un valor **int**.

Sintaxis:
```
#include<stdio.h>
int atoi(const char *cadena);
//Compatible ANSI, UNIX, Windows
```

atol convierte una cadena de carácter a un valor **long**

Sintaxis:
```
#include<stdio.h>
long atol(const char *cadena);
//Compatible ANSI, UNIX, Windows
```

sprintf

```
#include<stdlib.h>
int sprintf(char *buffer, const char *formato[,argumentos]…….);
//Compatibilidad ANSI, UNIX Y Windows
```

Esta función convierte los valores de los argumentos especificados a una cadena de carácter que almacena en el buffer. La cadena de carácter termina con el carácter nulo. Cada argumento es convertido y almacenado de acuerdo formato correspondiente que se haya especificado. Las especificaciones de los formatos son los mismos que para printf.

Ejemplo

```
#include<stdio.h>
#include<stdlib.h>
main()
{
char buffer[200] , s[  ]="ordenador", c='/';
int i=40, j;
float f=1.414214F;
j=sprinft(buffer+j,    "\t Cadena:                %s \n",s);
j=sprinft(buffer+j,    "\t Caracter:               %c \n",c);
j=sprinft(buffer+j,    "\t entero:               %d \n",i);
j=sprinft(buffer+j,    "\t float:               %f \n",f);
printf("Salida:\n%s\n Numero de caracteres =%d\n",buffer,j);
}
Salida:
        Cadena:Ordenador
        Carácter:    /
        Entero:      40
        Real:    1.414214
        Numeros de caracteres=72
```

Funciones para conversión de caracteres

toascii esta función pone a 0 todos los bits de c, excepto los 7 bits de menor orden es decir convierte c en un carácter ASCII

Sintaxis:
```
#include<ctype.h>
int toascii(int c);
//Compatible con UNIX y Windows
```

tolower convierte c a un caracter en minuscule, si procede.

Sintaxis:
#include<ctype.h>
int tolower(int c);
//Compatible con ANSCII, UNIX y Windows

toupper convierte c a un caracter en mayuscula, si procede.

Sintaxis:
#include<ctype.h>
int toupper(int c);
//Compatible con ANSCII, UNIX y Windows

Tipo y tamaño de una matriz

Usaremos typedef para obtener un sinónimo de un tipo de matriz como sigue:
typedef double t_matriz_1 d[100];

aqui se define un nuevo tipo, t_matriz_1 d se define una matriz unidimensional de 100 elementos de tipo double, asi se podrá definir matrices de este tipo ejemplo:

t_matriz_d m;

usando el operador sizeof podemos obtener el tamaño de la matriz.
Veamos
printf("No de elementos :%d\n", sizeof(m) / sizeof(m [0]));
otra forma puede ser:
printf("No de elementos :%d\n", sizeof(m) / sizeof(double));

Matrices multidimensionales

Estos usan más de un índice para referenciar los datos. Los arreglos multidimensionales son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arreglos más usuales son los de dos dimensiones, conocidos también por el nombre de tablas o matrices. Sin embargo, es posible crear arreglos de tantas dimensiones como requieran sus aplicaciones (programas), esto es, tres, cuatro o más dimensiones.

Matrices numéricas multidimensionales

La matriz numérica de varias dimensiones se crea de la siguiente forma:

Tipo nombre_matriz[expre_1][expre_2]….;

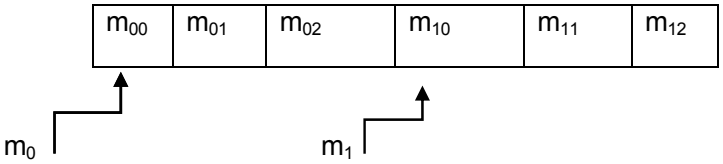
Tipo: es un número primitivo, entero o real.

El numero de elemento de una matriz multidimensional es el producto de las dimensiones indicadas en expre_1 y expre_2 etc.

Ejemplo

int m[2][3]; //las dimensiones de esta matriz es 2x3 =6;

C crea una matriz bidimensional m con dos elementos m[0] ,m[1] estas son dos matrices unidimensionales con tres elementos cada uno.



Los elementos de una matriz son colocados por filas consecutivas en memoria. El nombre de una matriz representa la dirección de la primera fila de la matriz o bien el primer elemento de la matriz asi:

m,m[0] y &m[0][0] son las misma dirección.

Desde nuestras perspectivas podemos ver la matriz como una tabla de f filas y c columnas.

Ejemplo

Matriz m

	Col 0	Col 1	Col 2
ila 0	m_{00}	m_{01}	m_{02}
ila 1	m_{10}	m_{11}	m_{12}

Los subíndices indica la posición del elemento, el primero la fila y el segundo índice la columna.

Veamos un ejemplo de matrices bidimensionales

```
#include<stdio.h>
#define Fil 3  //No de fila de la matriz
#define Col 3 //No de columna de la matriz
int main()
{
int i,j; //definimos los índice de la matriz para la fila y
columna
int m[Fil][Col]; //declaramos una matriz
//Leer los datos de la matriz 3 x 3
printf("\n Introd. loLos elementos de la Matriz")
for(i=0; i < Fil; i++)
 {
 for(j=0; j < Col; j++)
  {
  printf("Matriz[%i][%i]",i,j);
scanf("%d",&m[i][j]);
  }
}
```

```
//Visualizaremos los datos de la matriz

for(i=0; i < Fil; i++)
{
 printf("\n");
 for(j=0; j <Col; j++)
  printf("%d",m[i][j]);
}
}//fin del programa
```

Matrices de cadenas de caracteres

Las matrices de cadenas de caracteres son matrices multidimensionales, generalmente de dos dimensiones en la que cada fila se corresponde con una cadena de caracteres.

Se definen así:

Char nombre_matriz [filas] [longitud_fila];

Es una declaración como la siguiente char m[F][C]; F es el numero máximo de cadenas, y C es el numero máximo de caracteres de cada cadena.

Cuando se declara la matriz de tipo char de dos dimensiones como la anterior C crea una matriz unidimensional m con los elementos m[0], m[1], m[2], m[F-1], que a su vez es una matriz unidimensional de C elementos de tipo char.

Gráficamente nos imaginamos:

Podemos imaginarnos a esta matriz de cadenas de caracteres como una lista de cadenas.

m0
m1
.....

Cada una representa a una cadena, así que para acceder a la cadena de carácter solo necesitamos el primer subíndice, como ejemplo.

gets (m[0]); // accede a una cadena de caracteres

m[0] [c-1] = '\n'; // accede a un solo carácter de la cadena

Estructuras

Cuando se esta trabajando en C se declaran variables, que almacenan datos de un mismo tipo, como int var1 almacena un solo dato de tipo int; int a[7] almacena 7 datos de tipo int; pero no podemos guardar diferentes datos en esa misma variable. La finalidad de las estructura es agrupar una o mas variables, generalmente de diferentes tipos bajo un mismo nombre y hacer más fácil su manejo.

El caso mas típico de una estructura es una ficha de una persona en la que se encuentren todos sus datos nombres, apellidos, edad, dirección etc... Algunos de estos datos podrían ser estructura. Es como, si de un archivo se tratara.

Crear una estructura

La estructura es un nuevo tipo de dato que el usuario define, usando los tipos primitivos. Lo que nos indica que la declaración de un tipo de estructura, incluye tanto los elementos que la componen como sus tipos. Cada elemento de una estructura es conocido como miembro (o bien campo si se trata de registro).

Forma general:

```
struct tipo_estructura
{
   tipo miembro_1;
    tipo miembro_2;
…….
tipo miembro_n;
} [ lista de variables];
```

Ejemplo:

```
struct t_ficha
{
   char nombre [25];
   unsigned int edad;
    long teléfono;
   } var1, var2, var3;
```

Los miembros de una estructura no pueden tener los calificadores de clase de almacenamiento como: **extern, auto, static, register**

La estructura en C solo pueden contener miembros que se corresponden con definiciones de variables, C++ puede contener miembros que sean definiciones de funciones.

Definir una variable de tipo estructura

De forma análoga que las declaraciones de variables de tipo int, float, etc. También se pueden declara variables de tipo estructura.

En forma general así:

struct nombre_estructura [variable],[variable]….[variable];

Ejemplo

```
struct t_ficha  var1, var2, var3;      // correcto
```

No olvidar anteponer la palabra struct es decir no es valido lo siguiente

```
t_ficha  var1, var2, var3         // error
```

Podemos usar typedef para obtener un sinónimo de una estructura así

```
struct t_ficha
{
   char nombre [25];
unsigned int edad;
    long teléfono;
} ;
typedef struct t_ficha  Ficha;
```

Ahora ya se puede usar solo la Ficha así

```
Ficha var1, var2, var3;      // correcto
```

Acceso a los miembros de una estructura

Un miembro de una estructura, se utiliza exactamente igual que cualquier otra variable. Para acceder a sus miembros estructura se usa la notación siguiente:

variable_estructura.miembro

Ejemplo:

var1.telefono=25722079;

gets (var1.nombre);

Igual para var2, var3 y todas las variables que se declaren con iguales datos o distintos.

En ANSI C, el identificador de una estructura no comparte el lugar de almacenamiento del resto de los identificadores, y el nombre de un miembro de una estructura es local a la misma, y se utiliza solo después del operador(.) o (→), en caso d punteros.

Ejemplo:

```
#include<stdio.h>
typedef struct Ficha
{
char nombre [40];
   char direccion[40];
    long teléfono;
   } tficha;

int  Ficha =1;

main( )
{
tficha var1;
char nombre[40] = “Sofia” ;
printf(“nombre:  ” );
gets (var1.nombre);
printf(“%s \n” , var1.nombre);
printf((“%s \n” , nombre);
printf(“%d \n” , Ficha);
}
```

Se pueden usar nombres e identificadores iguales como Ficha en la estructura y Ficha variable entera, así como array miembro de la estructura y nombre un array declarado en la función main.

Si se ejecuta el programa pasando con gets Lola el resultado sería:

Nombre: Lola

Sofia

1

No es muy aconsejable usar identificadores de igual nombre pueden causar confusión.

Miembros que son estructuras

Podemos realizar estructuras que contengan miembros que son estructuras, estas se tiene que haber declarado previamente. Una estructura no puede contener un miembro d ella misma, pero si un puntero a un objeto de un mismo tipo.

Ejemplo:

```
struct fecha
{
   Int dia, mes, año;
   }
struct Ficha
{
     char nombre [40];
 char direccion[40];
    long teléfono;
   struct fecha  fecha_nacimiento;
} ;
struct  Ficha persona;
```

En esta estructura para accede al dato año de persona se escribiría así:

```
   Persona.fecha_nacimiento.año =1955;
```

Persona: variable de struct Ficha

Fecha_nacimiento: Miembro **de struct Ficha** y variable de tipo struct fecha

Así se accederá a todos los miembros de **struct fecha** que sean miembros de struct Ficha.

Con las estructuras podremos realizar las siguientes operaciones:

a) Iniciarla en el momento de definirla.
b) Obtener su dirección mediante operador &.
c) Acceder a uno de sus miembros.
d) Asignar una estructura a otra utilizando el operador de asignación.

Array de estructuras

Los arrays también pueden ser de tipo estructura, puesto que una estructura es un tipo de dato creado por el usuario, estos array de estructura se le denominan Matrices de estructura o bien d registros.

Para definir una matriz de estructura, primero hay que declarar un tipo de estructura que coincida con el tipo de elementos de la matriz.

```
typedef struct
{
char nombre [35];
   float nota;
   unsigned int curso;
} Matricula;
```

Se define la matriz:

```
Matricula  alumno[`50];
```

En el ejemplo se esta definiendo una matriz de 50 elementos de tipo estructura Matricula (alumno[0], alumno[1],…….alumno[99]). Para acceder a los miembros de la estructura nota, nombre, y curso, del alumno iésimo se utilizara la notación siguiente:

```
alumno[i].nombre;

alumno[i].nota;

alumno[i].curso;
```

UNIONES

Las uniones son en principio, entidades muy similares a las estructuras, están formadas por un número cualquiera de miembros, al igual que las estructuras, pero en este caso no existen simultáneamente todos los miembros, y solo uno de ellos tendrá un valor valido.

Una unión es una región compartida por dos o más miembros generalmente de diferentes tipos. Esto permite manipular distintos tipos de datos utilizando la misma zona de memoria, la reservada para la variable unión.

La declaración de una unión tiene la misma forma que la de la estructura, sustituyendo la palabra clave **struct** por **unión**. Todo lo que se planteo en la definición de una estructura vale para la unión, con la diferencia que los miembros de la unión no tienen cada uno su propio espacio de memoria como la estructura, los miembros de la unión comparten un único espacio de tamaño igual al del miembro de mayor longitud en bytes.

Forma general:

```
union  tipo_union
{
    /*declaración de los miembros*/
} ;
```

tipo_union es un identificador que nombra al nuevo tipo definido.

Se puede declarar variables de tipo unión igual que para estructura

unión tipo_union[variable [‘ variable…..];

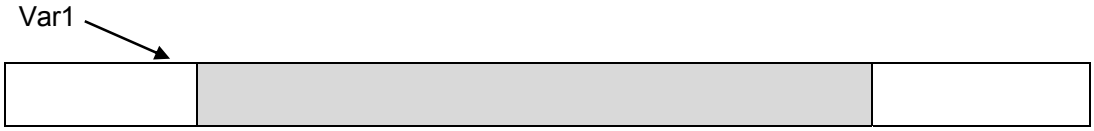
Una justificación de la existencia de la unión es la siguiente: si tenemos que guardar datos e inventario de materiales, los cuales algunos se identifican con su código (entero) y otros por su nombre (cadena de carácter), no tiene sentido definir dos variables, una para el entero y otra para la cadena de caracteres para cada articulo, si solamente se usa una forma u otra. Es por eso que las uniones resuelven este caso.

Para acceder a los miembros de una unión se utiliza los miembros operadores que los de la estructura.

```
variable_union.miembro;
Ejemplo:
union tmes
{
    char cmes [12];
int cmes;
float temperatura;
} ;
Declaración

union tmes var1;
```

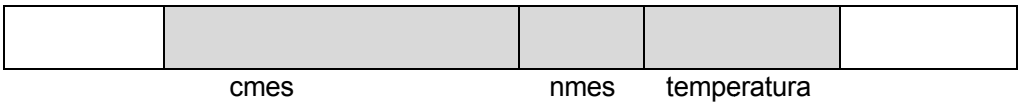
Para la variable var1 como los miembros union no tienen cada uno su zona de memoria, sino que comparten, el tamaño en bytes de var1 de tipo union tmes es de 12 bytes, que es la longitud de cmes miembro de mayor tamaño de la union. En forma nos imaginamos así:



almacena cmes o nmes, o bien temperatura

Se tiene que hacer notar que en una union solo uno de sus miembros puede estar almacenado y no todos a la vez como en la estructura, cada vez que se utiliza una variable union el valor almacenado es sobre escrito.

Si fuese una estructura gráficamente nos imaginamos de esta forma:



Campos de bits

A diferencia de otros lenguajes de programación, C tiene un método incorporado, denominado **campos bits** que permite acceder a los bits individuales. Los campos de bits son útiles por varias razones**:**

- Si el almacenamiento es limitado se puede almacenar varias variables boleanas (cierto/falso) en un byte.
- Ciertos dispositivos transmiten la información codificada en bits dentro de bytes.
- Ciertas rutinas de cifrado necesitan acceder a los bits de un byte.

Todas estas tareas se pueden realizar usando los operadores a nivel de bits, un campo de bits agiliza la estructura (y tal ves la eficacia) del código. Un campo de bits puede ser miembro de una estructura. Define en bits la longitud del campo.

Forma general:

Tipo nombre longitud:

Tipo: especifica el tipo de campo de bits.

Longitud: especifica el número de bits del campo.

Los tipos deben ser **int** o **unsigned int** (ANSI C). Los campos de bits se utilizan frecuentemente para analizar cualquier dispositivo hardware.

Ejemplo el puerto de estado de un adaptador de comunicación serie puede devolver el byte de estado así:

Bit	significado
0	Cambio de las líneas lista a enviar
1	Cambio de datos_listo
2	Direccion final
3	Cambio de línea recepción
4	Listo para enviar(CTS)
5	Datos listo a(DSR)
6	Llamadas telefónicas

Se puede representar esto usando una estructura de esta forma:

```
struct tipo_estado
{
unsigned delta_cts: 1;
unsigned delta_dsr: 1;
    unsigned tr_final: 1;
    unsigned delta_rec: 1;
unsigned cts:           1;
unsigned dsr:           1;
    unsigned ring:      1;
   unsigned linea:      1;
} estado;
```

Se podría trabajar así
estado =obtener_estado_Puerto( );

if(estado.cts) printf("Listo para enviar");

if(estado.dsr) printf("Dtos listos");

Se accede a los elementos con el operador (.)

estado.ring = 0 ; // se borra el campo ring

No es necesario nombrar cada campo de bits esto facilita llegar al bit que se esta usando pasando por alto los que se ocupan.

```
struct  tipo_estado
{
unsigned       :       4;
unsigned cts   :       1;
unsigned dsr   :       1;
} estado;
```

No se necesita especificar los bits que siguen a dsr si no se usan.

Los campos de bits tienen sus restricciones:

- No se puede tomar la dirección de una variable de campo de bits.
- No se pueden construir arrays de variables de campos de bits.
- La asignación de los campos de bits dependen del hardware.

TEMA VI: PUNTEROS

OBJETIVOS

CONTENIDO

1. Introducción
2. Concepto Básicos
3. Declaración de un Puntero
4. Paso de Puntero a una función.
5. Asignación dinámica de memoria

BIBLIOGRAFIA BASICA

Caballos Sierra Fco. Javier C/C++, CURSO DE PROGRAMACION (Texto Base) 2 da. Edicion Editorial RA-MA España 2002 ISBN 84-480—6

Deitel – P.J Deitel. **COMO PROGRAMAR EN C/C++**. H.M. 2 da. Edicion Editorial Prentine Hall Hispanoamericana, S.A. Mexico-ENGLEWOOD CLUFFS-Londres SYDNEY

Herbert Schild C Manual de Referencia 4ta. Edicion Editorial McGRAW-HILL/Osborne McGraw-Hill

Introducción

Al valor de cada variable se le asigna un lugar determinado en memoria, caracterizado por su dirección, el ordenador mantiene una tabla de direcciones que se relaciona con cada variable, por lo cual no tenemos que preocuparnos donde se encuentra registrado ese dato. C proporciona un operador (&) que nos permite determinar la dirección de la variable, mediante un tipo especial que contiene la dirección de esa. El cual se denomina puntero..

Puntero: Es una variable que contiene en su interior una dirección de memoria donde reside un dato. Es decir que el puntero apunta al espacio físico donde esta el dato o variable. Normalmente un puntero tiene un tipo de dato asociado.

.
Los punteros se pueden utilizar para:

- Referenciar y manipular estructura de datos
- Paso de argumento por referencia a una función
- Referencia de bloques de memoria asignación dinámica de memoria

Declaración de un puntero

Un puntero es una variable que almacena la dirección de memoria de otro objeto. O una variable que contiene en su interior una dirección de memoria donde reside un dato. Este dato “apuntado” por un puntero puede ser de cualquier tipo básico (char, short, int, long, float, double) o derivado (array, struct, union,…).

La sintaxis para declara un puntero es la siguiente:

<Tipo><variable puntero>

Tipo: tipo dato del objeto referenciado por el puntero.

Variable puntero: identificador de la variable tipo puntero.

Cuando se declara un puntero se reserva memoria para albergar una dirección de memoria, pero no para almacenar el dato al que apunta el puntero.

El espacio de memoria reservado para almacenar un puntero es el mismo independientemente del tipo de dato al que apunte: el espacio que ocupa una dirección de memoria.

```
char c='a';
char * pt;
int *  pr;
pt  =&c;
```

Dirección Memoria

Espacio reservado

Dirección	Contenido	Variable
1001		
1002	?	pr
1003	1004	pt
1004	a	c
1005		

Existe una dirección especial que se representa por medio de la constante NULL y se emplea cuando queremos indicar que un puntero no apunta a ninguna dirección.

Operadores * (indirección**)y &**(dirección)

Operador unitario (&). Denominado operador de dirección que permite hallar la dirección de la variable a que apunta. Un puntero.

&<id> devuelve la dirección de memoria donde comienza la variable <id>.

El operador & se utiliza para asignar valores a datos de tipo puntero:

```
int i;
int *ptr;
ptr = &i;
```

Operador unitario (*) que se denomina operador de indireccion el cual accede al valor depositado en la zona de memoria a la que apunta el puntero.

*<ptr> devuelve el contenido del objeto referenciado por el puntero <ptr>.

El operador * se usa para acceder a los objetos a los que apunta un puntero:

```
char c;
char *ptr;
ptr = &c;
*ptr = 'A'; // Equivale a escribir: c = 'A'
```

Operaciones con puntero

Asignación o Copiar un puntero en otro.

Ejemplo: main()
{
 int a=10, *p, *q;
p = &a; //se le asigna a p la dirección de a
q = p; //se le asigna a q la dirección de a

Sumar o restar a un puntero un número entero (útil sólo al apuntar a un array de datos).

Ejemplo: main()
{
 int x[100], *p=&x[3];
p = p+3; //p apuntará a x[6]
p = p-2;//p apuntará a x[4]

Incrementar (++) o decrementar (--) un puntero (útil sólo al apuntar a un array de datos).

Ejemplo: main()
{
int x[100], *p=&x[3];
p++; ++p; **//** p apuntará a x[5] ambos casos equivalen a p = p+1;
p--; --p; //p volverá a apuntar a x[3] ambos casos equivalen a p = p-1;

Restar entre sí dos punteros de un mismo tipo (útil sólo al apuntar a un array de datos).

Ejemplo:

 main()
{

int x[100], n;
int *p=&x[3], *q=&x[0];
n = p-q; **//**n valdrá 3 (distancia en datos int entre **p** y **q**)
n = q-p; **//**n valdrá -3 (distancia en datos int entre **q** y **p**)
Para saber la distancia en nº de bytes entre p y q:
n = (char *)p – (char *)q; **//**n valdrá 3x4=12 (suponiendo datos int de 4 bytes)

}

Comparación de punteros

La comparación de dos punteros es la comparación de dos enteros pues los punteros son enteros, la direcciones se corresponden con valores enteros, la ***comparación de punteros tiene sentido si estos apuntan a elementos del mismo array.***

```
int n = 10, *p = NULL, *q = NULL, x[100];
// ...
p = &x[99];
q = &x[0];
// ...
if (q + n <= p)
  q += n;
if (q != NULL && q <= p) // NULL es una constante que identifica
  q++;                   // a un puntero nulo
```

Puntero Generico: Un puntero "genérico" es aquel que puede apuntar a diversos tipos de datos a lo largo de la ejecución del programa. Se declara de la siguiente forma:

void *NombrePuntero

Una vez declarado, se puede copiar en él cualquier tipo de dirección, con un simple
operador de asignación (=): **NombrePuntero = &dato;**
Una vez cargado, para acceder al dato apuntado, hay que utilizar el operador de "indirección" (*), pero haciendo uso de un operador **"cast"** de conversión explícita de
tipo para punteros, que indique el tipo de dato que está siendo apuntado actualmente
por el puntero: **n = *(tipoDato *)NombrePuntero**

```
//Ejemplo de puntero Generico
#include <stdio.h>
#define FILAS 2
#define COLS 3
void CopiarMatriz( void *dest, void *orig, int n );

main()
{
  int m1[FILAS][COLS] = {24, 30, 15, 45, 34, 7};
  int m2[FILAS][COLS], f, c;

    CopiarMatriz(m2, m1, sizeof(m1));
   for (f = 0; f < FILAS; f++)
  {
    for (c = 0; c < COLS; c++)
      printf("%d  ", m2[f][c]);
      printf("\n");
   }
```

```
}

void CopiarMatriz( void *dest, void *orig, int n )
{
  char *destino = dest;
  char *origen = orig;
  int i = 0;
    for (i = 0; i < n; i++)
      destino[i] = origen[i];
}
```

Puntero Constante

Un puntero precedido por **const**, hace que el objeto apuntado sea constante, no sucediendo así con el puntero.

Const int *p=&a; //el objeto es constante el puntero puede cambiar de dirección

Int * const p=&a; //el objeto constante no puede cambiar de dirección

const int * p=&a; //ni el puntero ni el objeto pueden cambiar.

Errores comunes sobre punteros

Asignar punteros de distinto tipo

```
int a = 10;
int *ptri = NULL;
double x = 5.0;
double *ptrf = NULL;
...
ptri = &a;
ptrf = &x;
ptrf = ptri; // ERROR
```

Utilizar punteros no inicializados

```
char *ptr;
*ptr = 'a'; // ERROR
```

Asignar valores a un puntero y no a la variable a la que apunta

```
int n;
int *ptr = &n;
ptr = 9; // ERROR
```

Intentar asignarle un valor al dato apuntado por un puntero cuando éste es NULL

```
int *ptr = NULL;
*ptr = 9; // ERROR
```

Paso de Puntero a una función

El nombre puro de una función (sin los paréntesis) representa para el lenguaje C la dirección de memoria de comienzo del código máquina de dicha función. Tal dirección puede ser almacenada en una variable de tipo "puntero a función", y a través de esta variable, se puede lanzar la ejecución de la función apuntada y pasarle argumentos de entrada.

La declaración de una variable "puntero a función" se realiza así:
TipoDevuelto (*NombrePuntero)( );

donde **TipoDevuelto** es el tipo de dato que devuelve la función a la que apuntará este puntero, y donde los paréntesis extra alrededor del nombre del puntero y finales son imprescindibles. Si entre los dos paréntesis finales no se pone nada, indica que la función apuntada puede tener cualquier cantidad y tipo de argumentos de entrada. Ejemplo:

int (*pf)( );
La función apuntada debe devolver un valor int, y puede tener cualquier cantidad y tipo de argumentos de entrada. Si entre dichos paréntesis finales se ponen tipos de datos separados por comas, indicará que la función apuntada debe tener forzosamente tales tipos y cantidad de argumentos de entrada, y en ese orden. Esto quita flexibilidad y posibilidades a la variable puntero.
Ejemplo:
int (*pf)(int, float);

La función apuntada debe tener dos argumentos de entrada de tipos int y float, y debe devolver un valor int.

Para cargar el puntero (una vez declarado) con la dirección de memoria de una función, basta con asignarle su nombre puro. Ejemplo: **pf = NombreFuncion;**
Una vez cargado el puntero, para lanzar la ejecución de la función apuntada y pasarle argumentos de entrada caben dos sintaxis posibles:
n = pf(argum1, argum2);
Equivale a: n = NombreFuncion(argum1, argum2);
n = (*pf)(argum1, argum2); También equivale a lo mismo de antes
Evidentemente es más cómoda la primera de las dos sintaxis.

Se puede declarar un array de punteros de este tipo, con la siguiente sintaxis:
TipoDevuelto (*NombrePuntero[NumElementos]) ( );

Ejemplo

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

double cuadrado(double);
double pot(double, double);

typedef double (*t_ptFuncion)();

main()
{
  // Definir una matriz de punteros a funciones de tipo t_ptFuncion
  t_ptFuncion ptFuncion[10];

  // Asignar las direcciones de las funciones cuadrado y pot
  ptFuncion[0] = cuadrado;
  ptFuncion[1] = pot;
  /* Otras asignaciones */

  // Llamar a las funciones referenciadas por la matriz
printf("%g\n", ptFuncion[0](12.0));
  printf("%g\n", ptFuncion[1](12.0, 2.0));
}

double cuadrado(double a)
{
  return (a * a);
}

double pot(double x, double y)
{
  return exp(y * log(x));
}
```

Asignación Dinámica de Memoria

C cuenta con dos métodos para almacenar información en memoria. El primero es el que asigna el compilador utilizando variables globales y locales. El segundo método es mediante la utilización de funciones de biblioteca para la asignación de memoria dinámica en tiempo de ejecución del programa.

Esta técnica permite crear nuevas variables de memoria o arrays de cualquier tipo en tiempo de ejecución del programa, sin necesidad de haberlas declarado de antemano en el código fuente. Sólo es necesario declarar una variable puntero para cada nueva variable o array de datos que queramos crear. Esto se consigue principalmente con las funciones **malloc** y **free.**

malloc: esta función permite asignar un bloque de memoria de nbytes estos bytes consecutivos en memoria para almacenar uno o mas objeto de un tipo cualquiera. Esta función un ***void*** (puntero genérico) que referencia el espacio asignado.

La sintaxis es:

```
#include<stdlib.h>
void *malloc (size_t nbytes);
//compatible con ANSI, UNIX, Windows.
```

Si con la función malloc reservamos espacio para varios datos sucesivos en la memoria, estaremos creando un **"array dinámico"**, cuyo tamaño puede ser variado a lo largo de la ejecución del programa. Tal array será manipulado con facilidad mediante el puntero.

Existe otra función específica **calloc** para creación de arrays dinámicos, y la función **realloc** para modificación del tamaño en bytes de un array dinámico:

calloc

Esta función asigna memoria dinámica igual que **malloc** su sintaxis es:

```
#include<stdlib.h>
void *calloc(unsigned int NumElementos, unsigned int TamElemento);
//compatible con ANSI, UNIX, Windows
```

La diferencia entre las dos funciones son los parámetros, **calloc** recibe dos argumentos, el nº de elementos del array a crear, y el tamaño en bytes de uno de esos elementos. Reserva tantos bytes sucesivos como el producto de ambos argumentos, y además pone a ceros la zona de memoria recién reservada (cosa que no hace malloc).

En algunas ocasiones se necesita cambiar el tamaño de un bloque de memoria que se ha asignado previamente. Para tal razón existe una función de biblioteca de C denominada **realloc**.

Su sintaxis:

```
#include<stdlib.h>
void *realloc(void *Puntero, unsigned int Numbytes);
//compatible con ANSI, UNIX, Windows
```

La función **realloc** recibe dos argumentos, el puntero que apunta a la zona de memoria reservada ya existente, y el nuevo tamaño en nº de bytes deseado para tal zona. Los datos ya almacenados en la zona inicial no se pierden. Si el primer argumento (el puntero) vale NULL, realloc se comporta igual que malloc.

free

Esta función permite liberar la memoria que se ha asignado por las funciones **malloc, calloc o realloc**, pero no pone el puntero a NULL, si el puntero al que se le quiere liberar memoria esta a NULL esta función no hace nada.

Su sintaxis:
#include<stdlib.h>
void free(void *Puntero);
//compatible con ANSI, UNIX, Windows

Ejemplo de asignación dinámica de memoria

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

int main()
{
  int *p = NULL;
  int nbytes = 100;

  if ((p = (int *)malloc(nbytes)) == NULL )
{
    printf("Insuficiente espacio de memoria\n");
    exit(-1);
  }

  printf("Se han asignado %d bytes de memoria\n", nbytes);
  // ...
  free(p);
  return 0;
}
```

Arrays Dinámicos

Los arreglos con los que hemos trabajado hasta ahora han sido estáticos, por lo que es necesario conocer en el momento de escribir el código del programa, las dimensiones del arreglo y expresarla como una constante.

Los arreglos dinámicos utilizan técnica de asignación de memoria dinámica, es decir durante la ejecución del programa, este puede ser de cualquier tipo. Para crear un arreglo dinámico basta con declarar una variable que apunte a un objetos del tipo de los elementos del arreglo, e invocar a las funciones de asignación dinámica de memoria como **malloc, calloc o realloc**.

Veamos en siguiente trozo de programa como asigna y libera memoria dinamica

```
int *v=NULL //se declara e inicia un puntero a entero
int num=0;
printf("Numero de elementos de la vector");
scanf("%d",&num);
//Asignar memoria dinámica al vector
if ((v=(int*)malloc(num*sizeof(int))==NULL)
{
 printf("Insuficiente espacio de memoria");
exit(1);
}
//se inicia el vector
```

```
for (i=0; i<num ; i++)
v[i]=i;

//visualizar el vector

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

//Antes de finalizar el programa debemos liberar la memoria
free(v)
```

Arreglos dinámicos de dos dimensiones

Para asignar memoria dinámica a un arreglo de dos dimensiones, el proceso se divide en dos partes:

- Se asigna memoria para un arreglo de punteros, cuyos elementos referencia cada una de las filas de la matriz de dos dimensiones que se creara.
- Asignar memoria para cada una de las filas. El numero de elementos de la fila puede ser variable.

Veamos un ejemplo

```
//declarar un puntero a puntero Ejemplo
int **m
int nfil=0, ncols=0; //definimos las variables filas y columna para la matriz y se captura
                     //su valor por teclado
//asignación de memoria para la matriz de puntero
m=(int**)malloc(nfil*sizeof(int))

//notar que el tipo de m es (int **) y el tipo de m[i] es int*
//ahora asignaremos memoria para cada una de las filas de la matriz
//procedimiento de asignación memoria
for(i=0; f<nfil; i++)
{
if ((m[i]=(int*)malloc(ncol*sizeof(int))==NULL)
{
 printf("Insuficiente espacio de memoria");
 exit(1);
}
}


//proceso de liberación de memoria
for(i=0; i<nfil; i++)
free(m[i]); //libera memoria a cada una de las filas

//libera memoria para la matriz de puntero
free(m)
```

C cuenta con una función para iniciar la matriz a cero usando la función menset, que se encuentra definida en el fichero de cabecera memory.h

```
#include <memory.h>
//se usa la siguiente menset
for(i=0; f< filas; i++)
menset(m[i],0,ncol*sizeof(int));//inicializa a cero la matriz
```

Matrices Dinámicas de Cadenas de Caracteres

Esta es una cadena de carácter de dos dimensiones cuyos elementos son de tipo char, su construcción es igual a las matrices de dos dimensiones.

Como suelen ser de distintos tamaño se proceden a construir de la siguiente forma:

```
//Definimos la matriz puntero a puntero
char ** nombre;
//asignamos memoria dinámica a las filas
nombre=(char **)malloc (nfilas*sizeof(char *));
```

Como no se conoce el tamaño de la cadena este se almacena en un arreglo de caracteres, para conocer su longitud y asi asignar espacio para cada cadena, luego copiamos esta cadena en espacio reservado para ella.

Vemos el proceso:

```
f=0;
while(f<nfil && (longitud=strlen(gets(cadena)))>0)
{
 //asignamos espacio de memoria para cada cadena
 If( nombre[f]=(char*)malloc((logitud+1))==NULL)
 {
 printf("Insuficiente espacio de memoria");
 return(1);
 }
//Copiamos la cadena en el espacio de memoria
strcpy(nombre[f],cadena);
f++
}
```

Punterosa Estructuras

Se puede pasar una variable de tipo estructura o unión como argumento de entrada a una función también **"por referencia"**, es decir, que la función recibe la dirección de memoria donde reside la variable original. En la llamada a la función se utiliza el operador **"dirección de"** (**&**) para pasar la dirección de la variable estructura/unión.
En la cabecera de la función se recibe dicha dirección en una variable de tipo puntero a estructura/unión. La variable original sí puede ser alcanzada desde dentro de la función, a través de dicho puntero. Dentro de la función se utiliza el operador **"flecha"** (**->**) para alcanzar los campos internos de la variable estructura/unión original (**nombrePuntero->nombreCampo**).

Ejemplo

```
struct alumno
{
   char nombre[20];
    char apellido[20];
   flota nota;
}alumno1,alumno2;
```

Definimos **struct alumno** *pstruct;
pstruct=&alumno1; //le pasamos la dirección de la variable para poder acceder a los
                   //datos

Accedemos a los elementos de la estructura de la siguiente forma:
pstruct->nota o bien (*pstruct).nota

Además podemos asignarle memoria dinámica para reservar espacio.

pstruct = (struct alumno) malloc (sizeof (struct alumno));

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

struct fecha
{
  unsigned int dd;
  unsigned int mm;
  unsigned int aa;
};
```

```
void escribir(struct fecha *f);

int main()
{
  struct fecha *hoy; // hoy es un puntero a una estructura

  // Asignación de memoria para la estructura
hoy = (struct fecha *)malloc(sizeof(struct fecha));
if (hoy == NULL) return -1;

  printf("Introducir fecha (dd-mm-aa): ");
  scanf("%u-%u-%u", &hoy->dd, &hoy->mm, &hoy->aa);
  escribir(hoy);

  free(hoy);
  return 0;
}

void escribir(struct fecha *f)
{
  printf("Día %u del mes %u del año %u\n", f->dd, f->mm, f->aa);
}
```

A N E X O S

CLASE PRACTICA DE LABORATORIOS

Introducción

Conforme se va avanzando en la teoría es importante que el alumno vaya realizando ciertos ejercicios para llevar esa teoría a la práctica para ir afianzando y reforzando sus conocimientos adquiridos.

Además el alumno tiene que tener dominio del sistema operativo Windows, para poder crear archivos de textos, conocer y dominar la creación y manipulación de árbol de directorios, estos conocimientos y habilidades los deberá haber adquirido en la asignatura.

Para el laboratorio se cuenta con dos horas semanales, en las cuales, antes de iniciar una nueva practica, el profesor dará una guía para resolverla estando presente por si tiene duda o problema que se le presente a los estudiantes.

La asistencia al Laboratorio es obligatoria. Las practicas se realizan en grupos, el numero de alumnos por grupo va a estar en dependencia del numero de computadoras disponibles en el laboratorio, pero lo mas recomendable es que sean realizadas es individualmente.

La evaluación del laboratorio corresponde al 40% de las notas parciales de la asignatura. El estudiante deberá realizar todas las prácticas establecida en tiempo y forma para poder tener derecho al porcentaje del laboratorio. El estudiante deberá mostrar cada practica funcionando, se realizara una prueba escrita de 25% y entregar una memoria que contendrá la explicación de la resolución, el código comentado, conclusiones y comentarios con valor de 10%. La asistencia tiene un puntaje de 5%.

Planificación Temporal

Como ya se menciono con anterioridad, para el desarrollo de las prácticas de laboratorio se cuenta con una sesión de dos horas por semana para un total de 32 a lo largo de 16 semanas netas con la que se cuenta el laboratorio.

Buscando como exista concordancia entre la teoría y la practica, sin embargo, como muchas practicas no son posibles, se le va a facilitar al estudiante documentación con la información adicional necesaria (teoria, ejemplos, ayuda etc.) para la comprensión y desarrollo de las practicas.

Programación II

	Teoría	Practicas
1	Presentación de la Asignatura (1hr)	
2	Conceptos Básicos	
2		Practica No1 Entorno de desarrollo y compilación de un programa en C.
5	Operadores y Expresiones	
4		Practica No 2 Tipos de datos y Operadores en C
5	Sentencias de Control	
6		Practica No3 Sentencias de Control
5	Funciones	
4		Practica No 4 Funciones
6	Arreglos o Estructuras de Datos	
8		Practica No 5 Arreglos y Estructura de Datos
8	Punteros	
8		Practica No 6 Punteros

DESARROLLO DE LAS PRACTICAS DE LABORATORIOS

PRACTICA NO 1.

ENTORNO DE DESARROLLO Y COMPILACION DE UN PROGRAMA EN C

Esta primera práctica consiste en dar a conocer al alumno el entorno de trabajo, iniciando con los elementos básicos de programación, utilizando el entorno de Pelles C IDE para trabajar con programas escritos en C.

OBJETIVOS

- Conocer el entorno de desarrollo de Pelles C IDE
- Editar, compilar y ejecutar un programa sencillo
- Aprender a depurar un programa

HERRAMIENTAS NECESARIAS

Ordenador, Software Pelles C IDE

TEMPORIZACION

- Tiempo de Desarrollo de la practica: 1 semana
- Entrega de la práctica: Semana No 2.

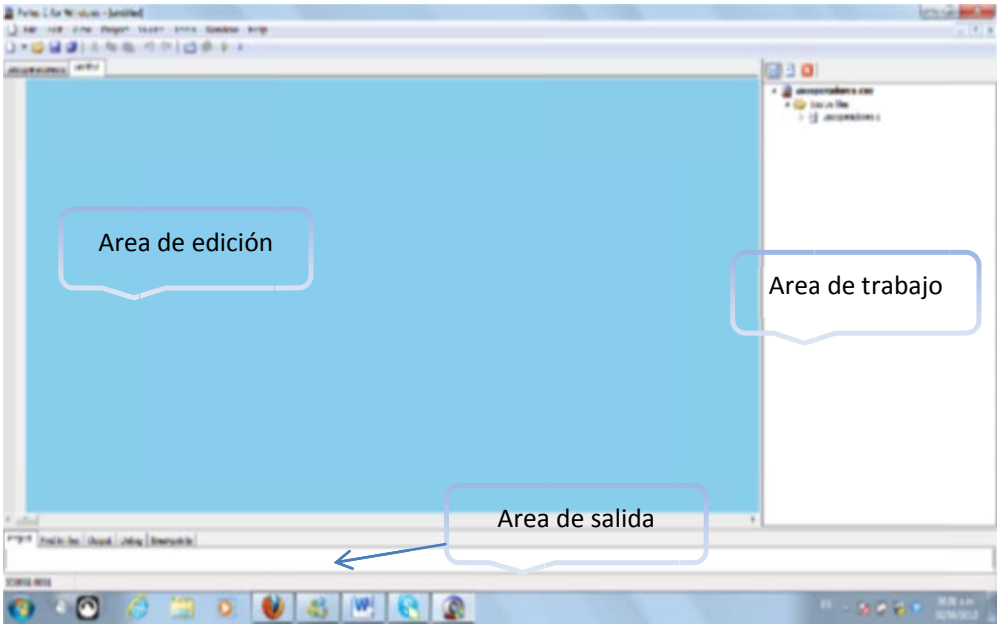
TAREA A REALIZAR

1 – Introducción a Pelles C IDE

Durante el curso emplearemos el entorno de desarrollo Pelles C IDE 6.0. Desde él es posible realizar todas las operaciones necesarias para desarrollar un programa en C, editarlo, compilarlo, enlazarlo, depurarlo y ejecutarlo. Además provee facilidades para gestionar varios proyectos compuestos por varios archivos fuentes, librerías, etc.

Para acceder podemos hacerlo de dos formas:

1. Pulsamos el botón "**inicio**" de Windows, y dentro del menú "**Todos los programas**", submenú "**Pelles C para Windows**" hacer clic en el icono "**Pelles C IDE**".
2. Otra forma haciendo clic en el icono que aparece en el escritorio de Windows "**Pelles C IDE**"



En la zona principal de la ventana de aplicación distinguimos tres áreas:

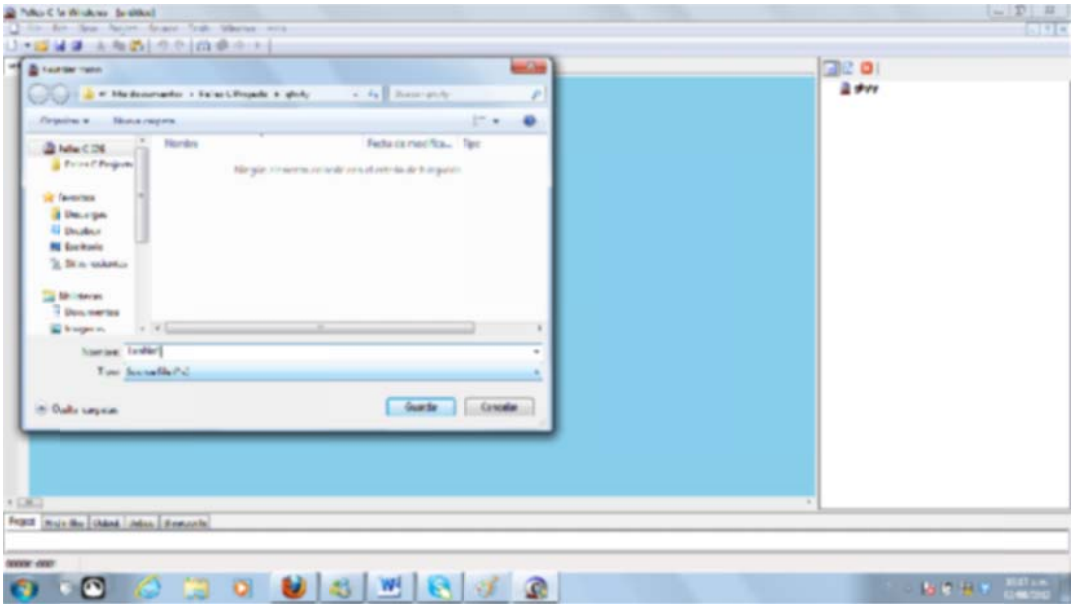
- Area de Edición: En ella se abrirá las ventanas que permitirán editar el código fuente de la aplicación.

- Area de salida: En esta ventana es donde aparecerán los mensajes de salida del compilador, el enlazador, depurador etc. Siempre serán mensajes informativos que nos indicaran si hay algún error al compilar el programa.

- Area de trabajo: En esta área aparecerá información diversa con la que estemos trabajando, como son los proyectos que tenemos abiertos, que ficheros fuente lo compone, y otras informaciones.

2 – Edición

Pulsamos en la barra de herramienta el botón **File – New – Project**, luego seleccionamos **Win32 console program**, le damos un nombre al proyecto y presionamos **OK.** Y abrimos una nueva ventana de edición **File – New – Source Code.** Seleccionamos la opción **File – Save** y le damos un nombre al archivo.



A continuación tecleamos el programa de prueba en la ventana de edición.

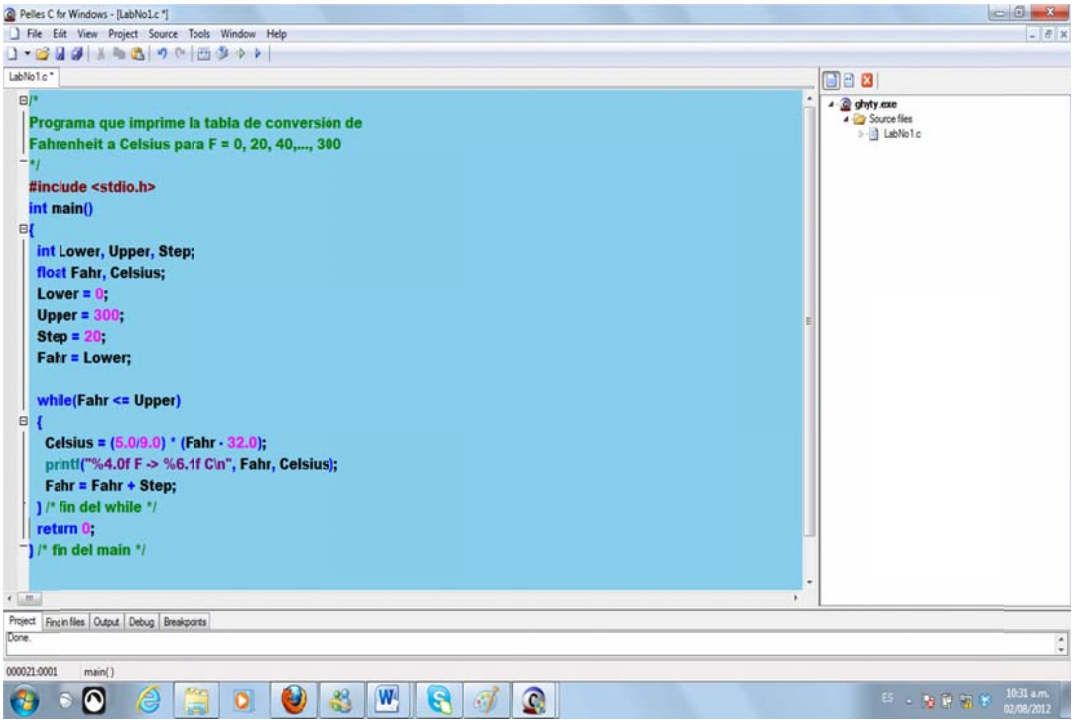
```
/*
Programa que imprime la tabla de conversión de
Fahrenheit a Celsius para F = 0, 20, 40,..., 300
*/
#include <stdio.h>

int main()
{
  int Lower, Upper, Step;
  float Fahr, Celsius;

  Lower = 0;
  Upper = 300;
  Step = 20;
  Fahr = Lower;

  while(Fahr <= Upper)
  {
    Celsius = (5.0/9.0) * (Fahr - 32.0);
    printf("%4.0f F -> %6.1f C\n", Fahr, Celsius);
    Fahr = Fahr + Step;
  } /* fin del while */

return 0;
} /* fin del main */
```



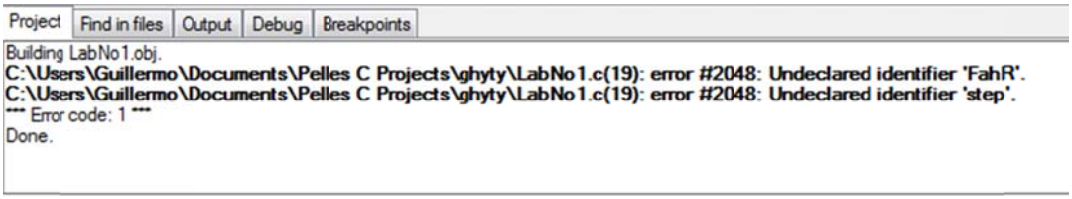
El código que hemos teclado nos muestra una tabla conversión de grados Fahrenheit a Celcius para F=0..300. No se preocupe por ahora si no entiende todos los detalles del mismo. Puede observar que el editor asigna colores a algunas palabras, observe también que el estilo de edición es importante; un código organizado ayuda a detectar errores y es mucho mas fácil de revisar.

Hemos introducido un error en el código, que veremos en el siguiente paso cuando compilemos el programa.

3 - Compilación y Enlazado

A continuación debemos compilar y enlazar el programa. Para ellos pulsamos control+B, alternativamente seleccionamos en el menú **Project** la opción **Build** o empleamos el botón de compilación de la barra de herramientas como podrá observar para la mayoría de funciones existe mas de una posibilidad.

Al intentar compile **Pelles C IDE** detectara que hay un error y detendrá la compilación, mostrando en la ventana de salida el error.



El mensaje indica en que línea de programa esta el error (línea 19) y de que tipo de error se trata. Nos indica que los identificadores FahR y step no esta definido.

hacemos doble clic en el área de salida, sobre la línea que indica en que línea de programa fuente se encuentra el error.

4 – Ejecución del Programa

Una vez corregido los errores ya se puede ejecutar el programa para ellos se debe pulsar la control + F5, de nuevo es posible usar el menú **Project**y la opción **execute** o el botón correspondiente de la barra de herramienta .

Como puede comprarse el programa funciona correctamente.

5 – Ejercicios a Realizar

Una vez que se halla familiarizado con el programa deberá resolver los siguientes ejercicios.

1. Escriba un programa que muestre por pantalla un mensaje que diga **Bienvenido a la Programación en Lenguaje C**.
2. Escriba un programa que calcule el área de un triángulo rectángulo, dada la altura y la base.
3. Escriba un programa que calcule los días vivido de una persona.

PRACTICA NO 2.

Tipos de Datos Y Operadores

OBJETIVOS

Conocer y utilizar los tipos de datos básicos y operadores en lenguaje C

TEMPORIZACION

Inicio de la práctica: Semana No 2

Tiempo de desarrollo de la práctica: 2 semanas

Finalización de la práctica: Semana No 4

Tarea a realizar

1 – Constantes, Variables y operadores

a) Operadores aritméticos: + - * / %

Realice un programa que dado dos números calcule la suma, resta, multiplicación, división y modulo de ambos. Según el siguiente algoritmo.

Pseudocódigo
Inicio
Leer x,y
z→x + y
visualizar z
z→x - y
visualizar z
z→x * y
visualizar z
z→x / y
visualizar z
z→x % y
visualizar z
fin del programa

b) Operadores de relación, lógicos y condicional: <><= >= != ==, && ||, ( )?:

Utilice los operadores relacionales y condicional para comprobar lo siguientes casos.
Ejemplo **z = (a < b) ? a : b;** si **a** menor que **b** entonces **a** de lo contrario **b**.
1) Si a= 70 y b= 2
2) Si a= 50 y b=30
3) Si a= 7 y b= 11
4) Si a= 100 y b= 50
5) Si a= 50 y b= 100

Ejercicios a Realizar

1. Realiza un programa que calcule el promedio de tres calificaciones.
2. Encuentre el mayor de dos números utilizando el operador condicional
   Ayuda mayor **= (**a**>**b**) ?** a **:** b**;**

PRACTICA NO 3.

Sentencias de Control

OBJETIVOS

Capturar y manejar los formatos de entrada y salida de C.
Utilizar correctamente lo que son las sentencias de control

TEMPORIZACION

Inicio de la práctica: Semana No 5

Tiempo de desarrollo de la práctica: 3 semanas

Finalización de la práctica: Semana No 8

Tarea a realizar

1 – Setencia de control **if – else.** Se desea saber si un número es par o impar. Veamos el siguiente programa en el cual el usuario, ingrese el número y el programa muestre con un mensaje, si éste es par o no.

```
# include < stdio.h >
int main( )
{
      int n;

   printf (" Ingrese el numero: ");
    scanf ("%d",&n);

     if ( (n % 2)==0)
        printf ("\n\n %d es un numero Par\n ",n);

     else
        printf ("\n\n %d es un numero Impar\n ",n);

    return 0;
}  // Fin del main
```

2- Sentencia **if** traduzca a Pelles el siguiente pseudocódigo

Pseudocodigo
Inicio
Leer numero
Si numero%2 entonces “el numero es múltiplo de 2” de lo contrario
Si numero%3 entonces “el numero es múltiplo de 3” de lo contrario
Si numero% 7 entonces “el numero es múltiplo de 7” de lo contrario
Si numero%13 entonces “el numero es múltiplo de 13”

3 – Digite el siguiente programa y compílelo utilizando switch luego realice los cambios conforme el ejercicio d.

```
#include <stdio.h>
main()
{
char tecla;
printf("Pulse una tecla y luego Intro: ");
scanf("%c", &tecla);
switch (tecla)
{
case ' ': printf("Espacio.\n");
break;
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
case '0': printf("Dígito.\n");
break;
default: printf("Ni espacio ni dígito.\n");
}
}
```

4 - Complete en el editor de Pelles C el siguiente trozo de programa para calcular la formula de la potencia $\mathbf{x^n}$ utilizando la sentencia **for**.

```
pot=1;
If (n==0)
  {
  printf("\n %d^%d=%d",base,n,pot)
  }
else
 {
for (i=1;i<n;i++)
 pot=pot*base;
printf("\n %d^%d=%d",base,n,pot)

}
```

5- Este programa Lee un texto desde el teclado mientras no se presione ctrl +z o fin de fichero. Haga las modificaciones correspondiente según ejercicios d y e.

```
#include<stdio.h>
int main()
{
char texto[80];
int cont, aux;
//Leer una linea de texto
for(cont=0;(texto[cont]=getchar())!='\n';++cont)
//apuntar al contador de caracteres
aux=cont;
//Escribir la línea de texto
for(cont=0;cont<=aux;++cont)
putchar(texto[cont]);
}
```

6 – El siguiente código de programa muestra por pantalla los múltiplos de 7 utilizando **do-while**

```
#include<stdio.h>
main()
{
int i=7;

do
{
printf(" %d",i);
i=i+7;
}while(i<100);

}//fin del programa
```

Ejercicios

a) En una tienda se venden artículos de primera necesidad, a los cuales se les aplica un descuento del 20%, de la compra total, si esta es igual o mayor a $50. Diseñe un programa en C, que a partir del importe total de la compra muestre lo que debe pagar el cliente.
b) Determinar la cantidad de dinero que recibirá un trabajador por concepto de las horas extras trabajadas en una empresa, sabiendo que cuando las horas de trabajo exceden de 40, el resto se consideran horas extras y que estas se pagan al doble de una hora normal cuando no exceden de 8; si las horas extras exceden de 8 se pagan las primeras 8 al doble de lo que se pagan las horas normales y el resto al triple. (salario se divide entre 30 días laborables para sacar lo que gana por día y luego se divide entre 8 para saber lo que gana por hora al día)
c) Encuentre el mayor de dos números utilizando el operador condicional
   Ayuda mayor **= (**a**>**b**) ?** a **:** b**;**
d) Transforme el programa a la sentencia **while** y visualice el texto leído en mayúscula.
e) Cuente la cantidad de vocales, dígitos, consonante, espacio en blanco y otros caracteres.

f) Elabore un programa que resuelva la siguiente formula 2^n utilizando la sentencia **while**
g) Mediante un bucle do-while mostrar todos los numero que sean múltiplos de 5 del 0 al 100.

PRACTICA NO 4.

Funciones en C

OBJETIVOS

Aprender a desarrollar funciones desarrollada por el usuario.
Dar conocer las ventajas que se obtienen en la utilización de función en un programa.

TEMPORIZACION

Inicio de la práctica: Semana No 9

Tiempo de desarrollo de la práctica: 2 semanas

Finalización de la práctica: Semana No 10

Tarea a realizar

1 –Edite el programa con funciones de usuario.

```
#include<stdio.h>
int suma (int a, int b); //función prototipo
main()
{
int x,y,z;
printf("Introd. Dos números:");
scanf("%d %d",&x,&y);
z=suma(x,y);//llamado a la funcion
printf("La suma de los dos números es:%d",z);
}
int suma (int a, int b)
{
return (a+b),
}
```

2 - Complete el siguiente trozo de programa en Pelles C IDE, este programa visualiza el mayor de dos numero.

```
main()
{
int a, b;
……
mayor(a,b);
……..
}


//Función mayor
int mayor (int x, int y)
{
return ((x>y)?x:y)
}
```

Ejercicios

a) Elabore una función que calcule el promedio de tres calificaciones.
b) Elabore una función que visualice los números pares del 1 al 50.
c) Elabore una función que compruebe si un carácter es digito, letras u otro tipo.
d) Elabore un programa de menú para las operaciones fundamentales utilizando funciones.

PRACTICA NO 5.

Arreglos

OBJETIVOS

Conocer y operar el manejo de los vectores, matrices y estructuras.
Conocer funciones que manipulan las cadenas de caracteres.
Saber definir los datos estructurados
Trabajar con arreglos estructurados.
.

TEMPORIZACION

Inicio de la práctica: Semana No 11

Tiempo de desarrollo de la práctica: 3 semanas

Finalización de la práctica: Semana No 13

Tarea a realizar

1 - Veamos el siguiente programa que pide saber el tamaño de una cadena.

```
#include<stdio.h>
#include<string.h>
main()
 {
char nombre[20];
int t;
printf("Introd. El nombre de una persona:");
gets(nombre);
t=strlen(nombre);//devuelve el tamaño de la cadena
printf("%s y sus tamaño %d ",nombre,t);
}
```

2 - Realice un programita que compare dos cadenas y diga si son iguales, si una es mayor o menor que la otra.

3 - Elabore un programa que lea el nombre y apellido de una persona los presente como una sola variable. Utilice la función de concatenación de cadena o unión de cadena.

4 - Compruebe, si el siguiente programa que calcula la edad promedio de n estudiante no tiene ningún error, en caso que tenga corregirlo y comentarlo.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int edad[100];//declaramos una variable tipo arreglo
int ind=1,n,sum=0;
float prom=0;
printf("Cuantos estudiantes a analizar");
scanf("%d",n);
while(ind<=n)
{
 printf("Ingrese la edad del estudiante");
scanf("%",&edad[ind]);
sum=sum++edad[ind];
ind++
}
prom=sum1/n;
printf("La edad Promedio es:%.2f",prom);
}
```

5 – Paso de vector a una función mediante el método de la burbuja, el cual consiste en ordenar una lista de números no ordenados.Además debe de corregir los errores que presente el programa y comentarlo.

```
// Método de la burbuja
#include <stdio.h>
#include <stdlib.h>

void ordenar(double m[], int n_elementos)
{
  // Ordenación numérica
  double aux;
  int i, s = 1;

  while (s && (--nelementos > 0))
{
    s = 0; // no permutación
for (i = 1; i <= n_elementos; i++)
// ¿ el elemento (i-1) es mayor que el (i) ?
if (m[i-1] > m[i])
     {
       // permutar los elementos (i-1) e (i)
```

```
aux = m[i-1];
      m[i-1] = m[i];
      m[i] = aux;
      s2 = 1; // permutación
     }
  }
}

void mostrar(double m[], int n)
{
  int i = 0;
  for (ie = 0; i < n; i++)
    printf("%g  ", m[i]);
  printf("\n");
}

main()
{
  double a[] = {5,  3,  1,  6,  2,  4}; //inicialización del arreglo
  int t = sizeof(a)/sizeof(double); //calculamos el tamaño del arreglo

  mostrar(a, t1);
  ordenar(a, t);
mostra(a, t);

  system("pause");
}
```

6- Arreglos Bidimensionales. El siguiente programa realiza la operación de suma de matrices pero debe corregir los errores del programa para que pueda funcionar.

```
#include<stdio.h>
#define Fil 3  //No de fila de la matriz
#define Col 3 //No de columna de la matriz
int main()
{
int i,j; //definimos los índice de la matriz para la fila y columna
int MA[Fil][Col],MB[Fil][Col],MC[Fil][Col]; //declaramos una matriz
//Leer los datos de la matriz 3 x 3
printf("\n Introd. loLos elementos de la Matriz A:\n")
for(i=0; i < Fil; i++)
 {
 for(j=0; j <Col; j++)
  {
  printf("Matriz[%i][%i]",i,j);
```

```
scanf("%d",&MA[i][j]);
}
}

printf("\n Introd. loLos elementos de la Matriz B:\n")
for(i=0; i < Fil; i++)
 {
 for(j=0; j < Col; j++)
  {
  printf("Matriz[%i][%i]",i,j);
scanf("%_",&M_[i][j]);
}
}
//Calcula la suma de la matrices
for(i=0; i < Fil; i++)
{
 for(j=0; j <Col; j++)
MC[i][j]=M_[i][j]+MB[i][j];
}

//Visualizaremos los datos de la matriz

for(i=0; i < Fil; i++)
{
 printf("\n");
 for(j=0; j <Col; j++)
  printf("%d",MC[i][j]);
}
}//fin del programa
```

7 – Complete el programa de estructura de datos y editelo. El siguiente programa visualiza el porcentaje de estudiante por departamento.

```
#include<stdio.h>
#include<stdlib.h>
#include<st___g.h>
struct estudiante
{
 char nombre[15];
 char procedencia[10];
 int edad;
}est[20];
int main()
{
        int i;
        int ct1=0,ct2=0,ct3=0,ct4=0,ct5=0; //inicializamos las variables contadoras
        //lectura de los datos
  printf("Introduzca los Datos de los estudiantes");
```

```
        for (i=0; i<5; i++)
        {
        printf("\nNombre:");gets(est[i].nombre);fflush(stdin);
        printf("Procedencia:");gets(est[i].procedencia);fflush(stdin);
                if(str__p(est[i].procedencia,"Leon")==0)
                {
                ct1++;
                }
                else
                if(st__mp(est[i].procedencia,"Managua")==0)
                {
                        ct2++;
                }
                else
                if(st__mp(est[i].procedencia,"Rivas")==0)
                {
                        ct3++;
                }
                else
                if(st__mp(est[i].procedencia,"Matagalpa")==0)
                {
                        ct4++;
                }

        printf("Edad:");scanf("%d",&est[i].edad);fflush(stdin);
        system("cls");
        }

//Viasualizacion de los datos de la estructura
        printf("\n");
        for(i=0; i<5; i++)
        printf("\n %s  %s   %d",est[i].nombre,est[i].procedencia,est[i].edad);

 printf("\nPorcentaje de estudiante por Municipio");
    printf("\n %d %.2f Leon ",ct1,(ct1/5.0)*100);
        printf("\n %d %.2f Managua ",ct2,(ct2/5.0)*100);
        printf("\n %d %.2f Rivas ",ct3,(ct3/5.0)*100);
        printf("\n %d  %.2f Matagalpa ",ct4,(ct4/5.0)*100);
}//fin del programa main
```

PRACTICA NO 6.

Punteros

OBJETIVOS

Conocer y utilizar los tipos de datos básicos y operadores en lenguaje C

TEMPORIZACION

Inicio de la práctica: Semana No 14

Tiempo de desarrollo de la práctica: 3 semanas

Finalización de la práctica: Semana No 16

Tarea a realizar

1 – Inicialización de un puntero mediante una variable tipo arreglo, el siguiente programa calcula el cuadrado del vector mediante la utilización de puntero.

```
#include<stdio.h>
int main()
{
int vector[]={2,3,4,5,6,7,8,9};//inicializamos el vector
int *pvect;//declamos un una variable tipo puntero.
int i,n;

 pvect=&vector[0];//le pasemos la direccion de inicio del vector a
la variable puntero.
 n=sizeof(vector)/sizeof(vector[0]);
//Calculamos el cuadro del vector mediante la variable puntero
      printf("Mostramos el cuadrado del vector\n");
      for(i=0; i<n; i ++)
      {
       printf(" %d",(*pvect+i)*(*pvect+i));
      }
putchar('\n');
}
```

2 – Asignación Dinámica de Memoria

Veamos el siguiente programa utilizando vectores dinámicos. Luego de escribir el código realizar la multiplicación de vectores.

```
//Suma de dos vectores mediante asignación dinámica de memoria
#include<stdio.h>
#include<stdlib.h>
int *asigna_memoria(int n)
{
  int *vector=NULL;
 vector=(int *)malloc(n*sizeof(int));
 if(vector==NULL)
          {
          printf("No se asignado memoria suficiente");
          exit(1);
          }
          return vector;
}

int main()
{
int *v1=NULL,*v2=NULL,*v3=NULL;
int i,n;

   printf("Introd. la cantidad de elmento de los vectores");
  scanf("%d",&n);
//Leemos el primer vector
          v1=asigna_memoria(n);
          for(i=0; i<n; i++)
          {
           printf("vector-1(%d)",i+1);fflush(stdin);
           scanf("%d",&v1[i]);
           }
          putchar('\n');
          v2=asigna_memoria(n);
          for(i=0; i<n; i++)
          {
           printf("vector-2(%d)",i+1);fflush(stdin);
           scanf("%d",&v2[i]);
           }
          //suma de vectores
          v3=asigna_memoria(n);
          for(i=0; i<n; i++)
          {
          *(v3+i)=*(v1+i)+(*(v2+i));
          }
          for(i=0; i<n; i++)
          {
```

```
        printf(" %d",*(v3+i));
        }
    //liberación de memoria
        free(v1);
        free(v2);
        free(v3);
}
```

3 – Veamos el siguiente ejemplo de asignación dinámica de memoria para matrices. Luego modifique el programa y realice la operación suma de matrices.

```
#include <stdio.h>
#include <stdlib.h>
int **asigna_memoria(int filas, int coln)
{
        int **matriz=NULL;
        int f;
 // Asignar memoria para la matriz de punteros
  if ((matriz = (int **)malloc(filas * sizeof(int *))) == NULL)
  {
    printf("Insuficiente espacio de memoria\n");
    exit (1);
  }

  // Asignar memoria para cada una de las filas
  for (f = 0; f < filas; f++)
  {
    if ((matriz[f] = (int *)malloc(coln * sizeof(int))) == NULL)
    {
      printf("Insuficiente espacio de memoria\n");
      exit (1);
    }
  }
        return matriz;
}
//Funcion Liberar memoria
int **liberar_memoria(int fil, int **matriz )
{
 int f;
// Liberar la memoria asignada a cada una de las filas
  for ( f = 0; f < fil; f++ )
    free(*(matriz+f));
  // Liberar la memoria asignada a la matriz de punteros
  free(matriz);
return matriz;
 }
int main()
{
  int **m = NULL;
  int nFilas = 0, nCols = 0;
  int correcto = 0, f = 0, c = 0;
```

```
  do
  {
    printf("Número de filas de la matriz:    ");
    correcto = scanf("%d", &nFilas);
    fflush(stdin);
  }
  while ( !correcto || nFilas < 1 );

  do
  {
    printf("Número de columnas de la matriz: ");
    correcto = scanf("%d", &nCols);
    fflush(stdin);
  }
  while ( !correcto || nCols < 1 );
//llamado a la funcion asigna_memoria
m=asigna_memoria(nFilas,nCols);

  // Iniciar la matriz a cero
  for ( f = 0; f < nFilas; f++ )
    for ( c = 0; c < nCols; c++ )
     (*(*(m+f)+c))=0;
     // m[f][c] = 0;

  //Leer la matriz
  printf("\n Introduzca los Elementos de la matriz\n");
  for ( f = 0; f < nFilas; f++ )
   for ( c = 0; c < nCols; c++ )
            {
   printf("Matriz[%i][%i]",f,c);fflush(stdin);
   scanf("%d",&(*(*(m+f)+c)));
              }

   // Visualizar la matriz 2D
  for ( f = 0; f < nFilas; f++ )
  {
    for ( c = 0; c < nCols; c++ )
      //printf("%d  ", m[f][c]);
     printf("%d  ", (*(*(m+f)+c)));
    printf("\n");
  }

liberar_memoria(nFilas,m);
 return 0;
}
```

4 -Asignación dinámica de memoria a estructura de datos.
Edite el siguiente programa en Pelles y corrija los errores correspondientes.

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

struct fecha
{
  unsigned int dd;
  unsigned int mm;
  unsigned int aa;
};

void escribir(struct fecha *f);

int main()
{
  struct fecha *hoy; // hoy es un puntero a una estructura

  // Asignación de memoria para la estructura
hoy = (struct fecha *)malloc(sizeof(struct fecha));
if (hoy == NULL) return -1;

  printf("Introducir fecha (dd-mm-aa): ");
  scanf("%u-%u-%u", &hoy->dd, &hoy->mm, &hoy->aa);
  escribir(hoy);

  free(hoy);
  return 0;
}

void escribir(struct fecha *f)
{
  printf("Día %u del mes %u del año %u\n", f->dd, f->mm, f->aa);
}
```