UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN
CENTRO DE EDUCACIÓN A DISTANCIA Y GESTIÓN DEL CONOCIMIENTO

APUNTE ELECTRÓNICO

Introducción a la Programación

Licenciatura en Informática

Plan 2012 2016 actualizado

COLABORADORES

DIRECTOR DE LA FCA
Mtro. Tomás Humberto Rubio Pérez

SECRETARIO GENERAL
Dr. Armando Tomé González

– – – –

COORDINACIÓN GENERAL
Mtra. Gabriela Montero Montiel
Jefa del Centro de Educación a Distancia y Gestión del Conocimiento

COORDINACIÓN ACADÉMICA
Mtro. Francisco Hernández Mendoza
FCA-UNAM

COORDINACIÓN DE MULTIMEDIOS
L.A. Heber Javier Mendez Grajeda
FCA-UNAM

– – –

COAUTOR
L.I. Espartaco David Kanagusico Hernández
L.C. Gilberto Manzano Peñaloza

REVISIÓN PEDAGÓGICA
Mayra Lilia Velasco Chacón

CORRECCIÓN DE ESTILO
Mtro. Francisco Vladimir Aceves Gaytán

DISEÑO DE PORTADAS
L.CG. Ricardo Alberto Báez Caballero

DISEÑO EDITORIAL
Mtra. Marlene Olga Ramírez Chavero

Dr. Enrique Luis Graue Wiechers
Rector

Dr. Leonardo Lomelí Vanegas
Secretario General

Mtro. Tomás Humberto Rubio Pérez
Director

Dr. Armando Tomé González
Secretario General

Mtra. Gabriela Montero Montiel
Jefa del Centro de Educación a Distancia
y Gestión del Conocimiento / FCA

Introducción a la programación
Apunte electrónico

Edición: agosto 2017

Facultad de Contaduría y Administración
Circuito Exterior s/n, Ciudad Universitaria
Delegación Coyoacán, C.P. 04510, México, Ciudad de México.

ISBN: En trámite
Plan de estudios 2012, actualizado 2016.

Hecho en México

OBJETIVO GENERAL

El alumno será capaz de implementar algoritmos en un lenguaje de programación.

TEMARIO OFICIAL

(Horas 64)

	Horas
1. Introducción a la programación	4
2. Tipos de datos elementales (variables, constantes, declaraciones y expresiones y estructura de un programa)	6
3. Control de flujo	14
4. Funciones	18
5. Tipos de datos compuestos (estructuras)	14
6. Manejo de apuntadores	8

INTRODUCCIÓN A LA ASIGNATURA

Las notas explican los puntos necesarios para el desarrollo de programas de computadora. Se tratan conceptos básicos y de la estructura de un programa, además de temas avanzados como son el uso de apuntadores y archivos.

Primera unidad

En la *primera unidad* (introducción a la programación) se mencionan conceptos básicos de programación.

Segunda unidad

En la *segunda unidad* (tipos de datos elementales: variables, constantes, declaraciones, expresiones y estructura de un programa) se enumeran dichos conceptos, que son los elementos que construyen un programa.

Tercera unidad

En la *tercera unidad* (control de flujo) se analiza la utilización de la estructura secuencial, condicional y repetitiva.

Cuarta unidad

En la *cuarta unidad* (funciones) se analiza la función, y su utilidad para la realización de tareas específicas dentro de un programa**.**

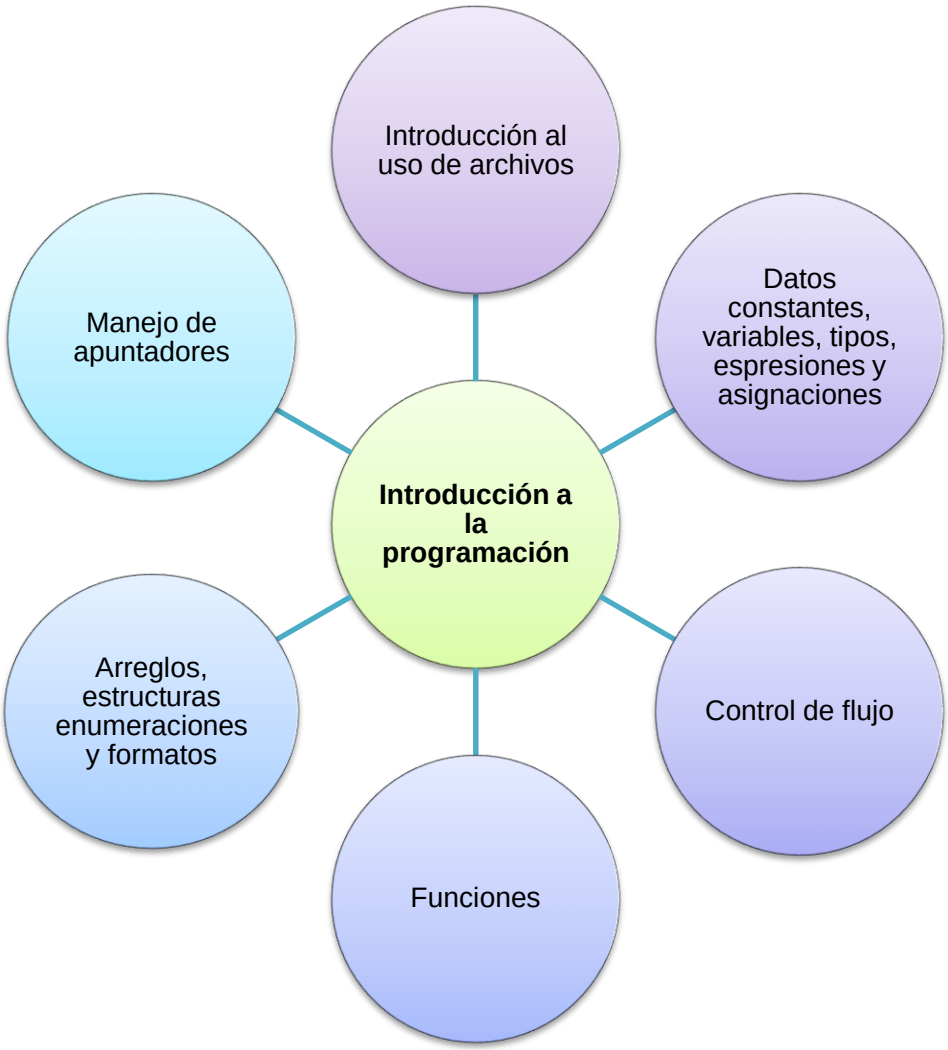
Quinta unidad

En la *quinta unidad* (tipos de datos compuestos: estructuras) se desarrollan programas que utilizan los arreglos y estructuras para almacenar y manipular datos de un solo tipo o diferentes**.**

Sexta unidad

En la *sexta unidad* (manejo de apuntadores) se utilizan los apuntadores para la realización de programas que utilizan memoria dinámica.

ESTRUCTURA CONCEPTUAL


Introducción al uso de archivos
Manejo de apuntadores
Datos constantes, variables, tipos, espresiones y asignaciones
Introducción a la programación
Arreglos, estructuras enumeraciones y formatos
Control de flujo
Funciones

UNIDAD 1

Introducción a la programación

OBJETIVO PARTICULAR

Será capaz establecer la diferencia entre los paradigmas de programación e identificar los lenguajes de acuerdo con su nivel y sus principales características.

TEMARIO DETALLADO

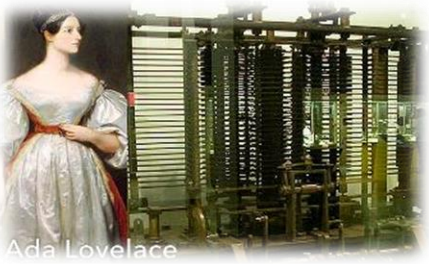
(4 horas)

1. Introducción a la programación

- 1.1. Concepto de lenguaje de programación
- 1.2. Paradigmas de programación
  - 1.2.1. Paradigma imperativo
  - 1.2.2. Paradigma orientado a objetos
  - 1.2.3. Paradigma funcional
- 1.3. Lenguaje máquina
- 1.4. Lenguajes de bajo nivel
- 1.5. Lenguajes de alto nivel
- 1.6. Intérpretes
- 1.7. Compiladores
- 1.8. Fases de la compilación
- 1.9. Notación BNF
- 1.10. Sintaxis, léxico, semántica

INTRODUCCIÓN

Los primeros lenguajes de programación surgieron de la idea de Charles Babagge, quien fue un profesor de la Universidad de Cambridge, a mediados del siglo XIX. Él es considerado como el precursor de muchas de las teorías en que se basan las computadoras.



Babagge diseñó una máquina analítica (la primera calculadora numérica universal), que por motivos técnicos no pudo construirse sino hasta mediados del siglo XX. En este proyecto colaboró directamente Ada Augusta Byron, (hija del poeta Lord Byron), quien es considerada como la primera programadora de la historia, y quien realizó programas para dicha máquina. Debido a que ésta no llegó a construirse, los programas de Ada tampoco llegaron a ejecutarse, pero sí suponen un punto de partida de la programación, sobre todo si observamos que los programadores que les sucedieron utilizaron las técnicas diseñadas por ella y Babagge, que consistían principalmente, en la programación mediante tarjetas perforadas.

Lenguaje de programación

Es una herramienta que permite desarrollar programas para computadora. Dichos lenguajes permiten expresar las instrucciones que el programador desea se ejecuten en la computadora.

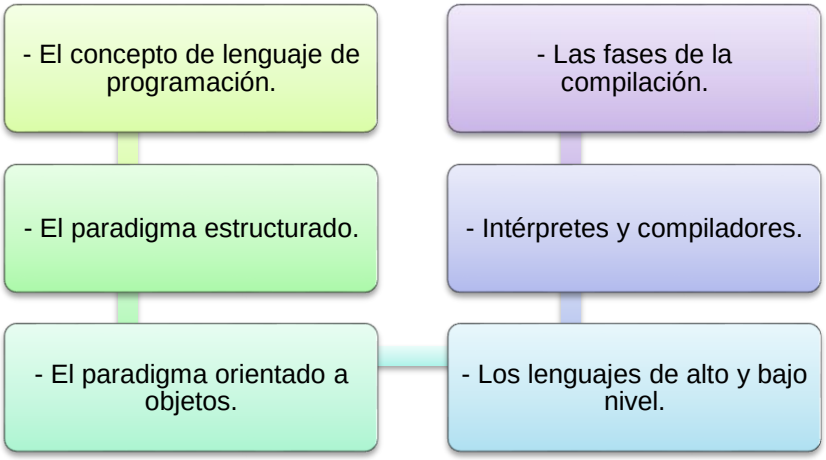
La función de los lenguajes de programación es realizar programas que permitan la resolución de problemas.

Existen, además, dos herramientas muy utilizadas para transformar las instrucciones de un lenguaje de programación a código máquina. Estas herramientas se denominan **intérpretes** y **compiladores**.

Intérpretes	Compiladores
•Los **intérpretes** leen las instrucciones línea por línea y obtienen el código máquina correspondiente.	•Los **compiladores** traducen los símbolos de un lenguaje de programación a su equivalente escrito en lenguaje de máquina. A ese proceso se le llama compilar. Por último se obtiene un programa ejecutable.

Para ampliar más la información referente a este tema, es recomendable que leas el documento anexo de **lenguajes de programación** (**ANEXO 1**).

En esta unidad se tratarán los temas introductorios de la programación de computadoras:



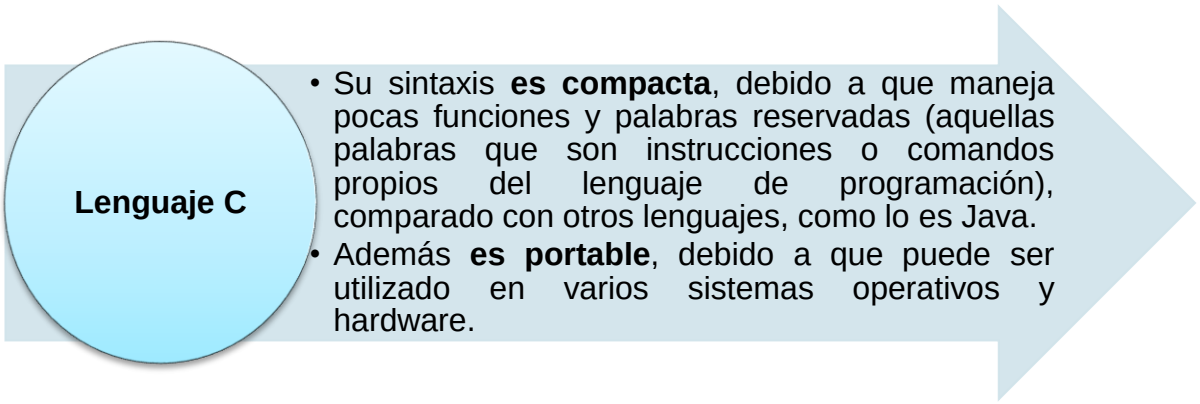
1.1. Concepto de lenguaje de programación

Un lenguaje de programación es una herramienta que permite desarrollar programas para computadora.

La función de los lenguajes de programación es escribir programas que permiten la comunicación usuario/máquina. Unos programas especiales (compiladores o intérpretes) convierten las instrucciones escritas en código fuente, a instrucciones escritas en lenguaje máquina (0 y 1).

Para entender mejor cómo se estructura un lenguaje de programación, estaremos empleando a modo de ejemplo el lenguaje C, que se caracteriza principalmente por ser de uso general, de sintaxis compacta y portable.

Decimos que el lenguaje C **es de uso general**, ya que puede usarse para desarrollar programas de diversa naturaleza como lenguajes de programación, manejadores de bases de datos o sistemas operativos.



1.2. Paradigmas de programación

Un paradigma es un modelo que, a su vez, es una representación abstracta de la realidad.

Un paradigma de programación es un modelo de programación que representa un estilo o forma de programar o construir programas para realizar ciertas tareas o actividades. Cada modelo tiene sus propias estructuras y reglas de construcción. El modelo de programación por emplear depende del problema que se desee solucionar.

1.2.1. Paradigma imperativo

La programación imperativa es una forma de escribir programas secuenciales; es decir, que tienes que ir indicando en el programa los pasos o tareas que debe realiza según las siguientes reglas:

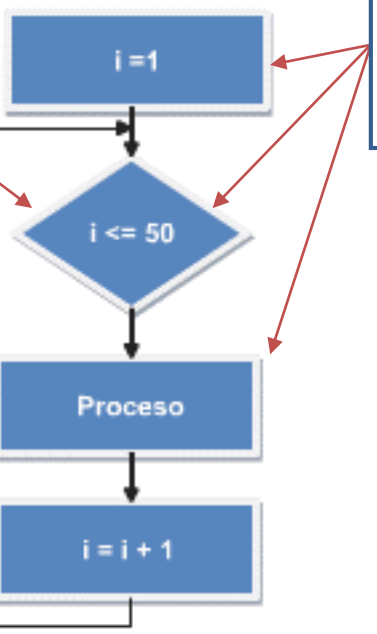
1. El programa tiene un diseño modular.

2. Los módulos son diseñados de manera que un problema complejo se divide en problemas más simples.

3. Cada módulo se codifica utilizando las tres estructuras de control básicas: secuencia, selección y repetición.

Estructura de selección. Consiste en una decisión que tiene dos alternativas: verdadero o falso.

Estructura secuencial. Es una instrucción que va seguida de otra.



Estructura de repetición. Es un ciclo en donde se ejecuta una o más instrucciones dependiendo de una condición.

Existen diversos lenguajes estructurados como Pascal y Fortran, así como también lo es C. Cabe destacar que en el lenguaje C ha sido desarrollada la mayoría de los sistemas operativos, manejadores de bases de datos y aplicaciones de la actualidad, esto se debe a que el código producido es muy eficiente: tiene estructuras de un lenguaje de alto nivel, pero con construcciones que permiten el control a un bajo nivel, e incluso mezclar su código con el lenguaje ensamblador para tener un acceso directo a los dispositivos

periféricos. Además, su código es elegante en su construcción por el grado de abstracción que permite, lo cual lo hace a la vez un lenguaje complejo para los programadores novatos.

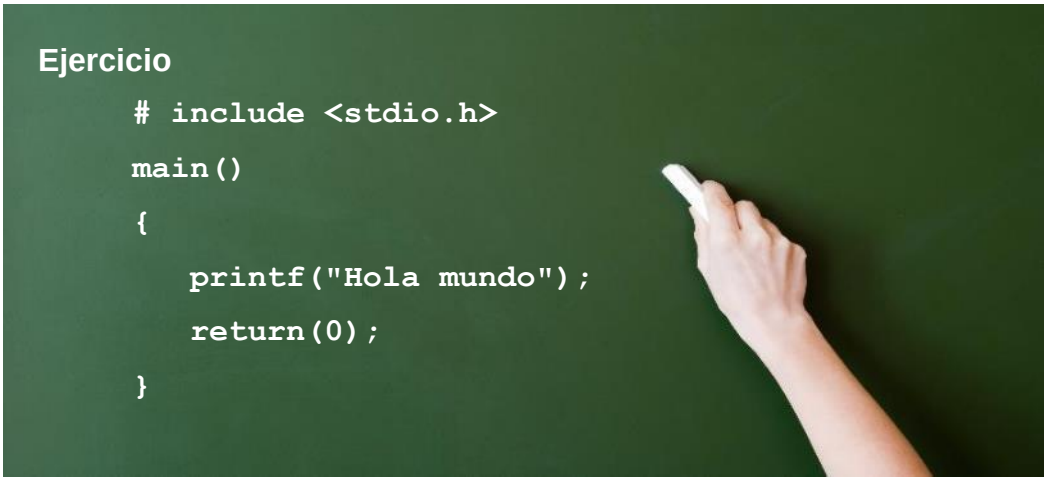
Estructura de un programa en C

Todos los programas en C consisten en una o más funciones, la única función que siempre debe estar presente es la denominada main(), siendo la primera función que se invoca cuando comienza la ejecución del programa.

Forma general de un programa en C:

Archivos de cabecera	/*Son bibliotecas de programas externos que contienen funciones preprogramadas y que tienen la extensión .h que es la inicial de Header o cabecera*/
Declaraciones globales	/*Son variables cuyo ámbito es tanto en el programa principal como en los programas que este ocupe, se declaran e inicializan al principio del programa principal*/
tipo_devuelto main(parámetros)	/*Los paréntesis son parte de la sintaxis de una función y encierran a los argumentos o parámetros que esta requiere para realizar su tarea*/
{ ……..	
sentencias(s);	/*En esta sección va el conjunto de instrucciones o sentencias que ocupa el programa*/
}	
tipo_devuelto función(parámetros)	
{ ……..	
sentencias(s);	
}	

Ejemplo de un programa en C



Este primer programa muestra en la pantalla de la computadora el mensaje “Hola mundo”.

La **primera línea** #include <stdio.h> se denomina *Encabezado*, y es un archivo que proporciona información al compilador. En este caso en particular, se invoca al archivo stdio.h que, entre otras funciones, incluye la función *printf* que es la que permite mostrar contenidos en pantalla, si no se utiliza este archivo la función de *printf* no podrá utilizarse en el programa. Cabe hacer notar que se utiliza el símbolo # para indicar al programa que se va a hacer referencia a una librería de funciones, este símbolo se pronuncia como “*pound*”. La palabra reservada include se usa para utilizar una librería de funciones específica.

La **segunda línea** del programa contiene el nombre de la función principal main(), cuya sintaxis es *nombre de función(parámetros o argumentos),* en este caso la función no ocupa argumentos, pero por razón de sintaxis la función debe llevar los paréntesis.

Como se aprecia en el código fuente, se abre una llave “{” la cual se cierra con su contraparte “}” al final del programa, esto indica el cuerpo del programa en donde van contenidas las instrucciones o sentencias que ocupa el mismo programa.

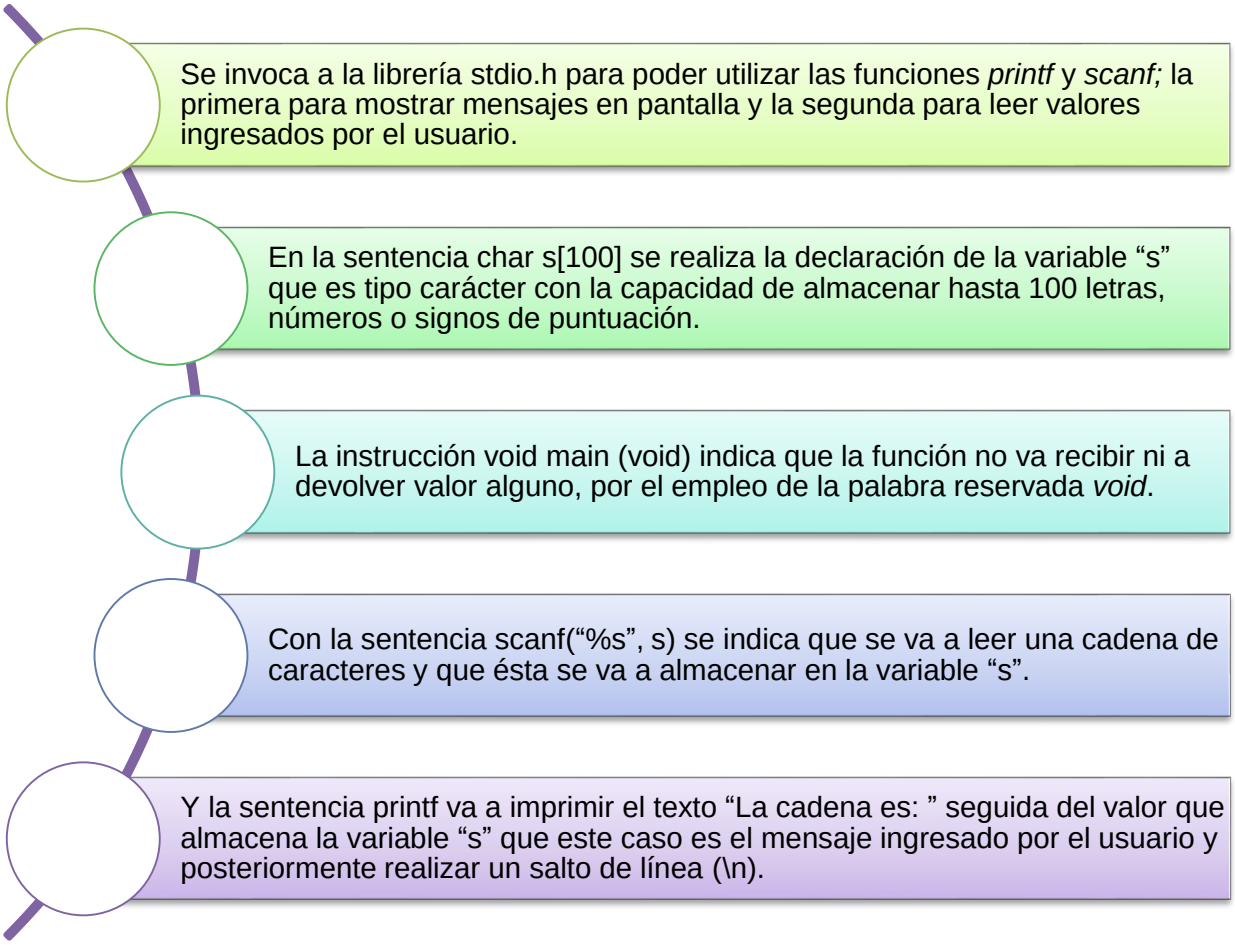
La única instrucción que contiene el programa es *printf,* que muestra el mensaje “Hola mundo” en la pantalla. Se aprecia que la frase va entrecomillada porque es texto y a la vez es el argumento de la función *printf*. Al final de cada línea debes cerrarla con un punto y coma “;” para indicar el fin de instrucción, puedes utilizar varias instrucciones en una sola línea.

Por último, la sentencia *return*(0) indica que esta función devuelve un cero, que es un valor que el programa principal no ocupa, pero por sintaxis toda función debe de devolver algún valor.

El siguiente ejercicio permite introducir el mensaje que será mostrado en pantalla:

```
#include <stdio.h>

char s[100];
void main(void)
{
printf("Introduzca el mensaje que desea desplegar\n");
scanf("%s",s);
printf("La cadena es: %s\n",s);
}
```

Este programa, al igual que el anterior, imprime un mensaje en pantalla, pero con la novedad de que dicho mensaje lo ingresa el usuario, a continuación, se explica cómo lo hace:



Dado que C es el lenguaje de programación con el que ha sido desarrollada la mayoría de los programas de cómputo actuales, es lógico pensar en el mismo como uno de los más importantes.

1.2.2. Paradigma orientado a objetos

Los conceptos de la programación orientada a objetos tienen origen en Simula|Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo. Al parecer, este centro trabajaba en simulaciones de naves, y fueron confundidos por la explosión combinatoria de cómo las distintas cualidades de varias naves podían afectarse unas a las otras. La idea ocurrió

para agrupar los diversos tipos de naves en otras clases de objetos, siendo responsable cada clase de objetos de definir sus "propios" datos y comportamiento. Fueron refinados más tarde en Smalltalk que fue desarrollado en Simula y cuya primera versión fue escrita sobre Basic, pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "en la marcha" en lugar de tener un sistema basado en programas estáticos.

Ole-Johan Dahl y Kristen Nygaard

La programación orientada a objetos tomó posición como el estilo de programación dominante a mediados de los años ochenta, en gran parte por la influencia de C++ (una extensión del lenguaje C). Su dominación fue consolidada gracias al auge de las interfaces gráficas, para las cuales la programación orientada a objetos está particularmente bien adaptada.

Las características de la orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal|, entre otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas, condujo frecuentemente a problemas de compatibilidad y a la capacidad de mantenimiento del código. Los lenguajes orientados a objetos "puros"; por otra parte, carecían de las características de las cuales muchos programadores habían venido a depender. Para saltar este obstáculo, se hicieron muchas pruebas para crear nuevos lenguajes basados en métodos orientados a objetos, pero añadiendo algunas características imperativas de manera "seguras". El lenguaje de programación Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos, pero ahora ha sido esencialmente reemplazado por Java, en gran parte debido a la aparición de Internet, y a la implementación de la máquina virtual de Java en la mayoría de los navegadores. PHP, en su versión 5.3.8 (23 de agosto de 2011), se ha ido modificando y soporta una orientación completa a objetos, cumpliendo todas las características propias de la orientación a objetos.

Las características más importantes de la programación orientada a objetos son las siguientes:

Abstracción (Román, 2002, §3.6)

Denota las características esenciales de un objeto, donde se captura su comportamiento. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar "cómo" se implementan estas características. Los procesos, las funciones o los métodos, pueden también ser abstraídos, y cuando sucede esto, una variedad de técnicas son requeridas para ampliar una abstracción.

Encapsulamiento (Román, 2002, §3.3)

Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

Principio de ocultación

Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una "interfaz" a otros objetos, que específica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

Polimorfismo (Román, 2002, §3.5)

Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. Dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama "asignación tardía" o "asignación dinámica". Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la [[sobrecarga| sobrecarga de operadores]] de C++.

Herencia (Román, 2002, §3.4)

Las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en "clases" y estas en "árboles" o "enrejados" que reflejan un

comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay "herencia múltiple".

Recolección de basura
La recolección de basura o *Garbage Collection* es la técnica por la cual el ambiente de Objetos se encarga de destruir automáticamente, y por tanto, desasignar de la memoria los Objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo Objeto y la liberará cuando nadie lo esté usando. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse manualmente.

1.2.3. Paradigma funcional

La programación funcional tiene como objeto imitar las funciones matemáticas lo más posible. Posee la propiedad matemática de transparencia referencial, lo que significa que una expresión representa siempre el mismo valor, permitiendo razonar sobre la ejecución de un programa y demostrar matemáticamente que es correcto.

Las variables son como las variables en algebra, inicialmente representan un valor desconocido que, una vez calculado, ya no cambia.

El orden de evaluación de las sub expresiones no afecta al resultado final, por lo tanto, las sub expresiones pueden ejecutarse en forma paralela para hacer más eficiente el programa.

Cuando se aplica una función, los argumentos que ésta toma pueden ser:

Evaluados antes de llamar la función (evaluación estricta).

Evaluados dentro de la función hasta el último momento y solo si se requieren para calcular el resultado final (evaluación postergada).

Bases de la programación funcional

Las funciones matemáticas son una correspondencia entre un dominio y un rango.

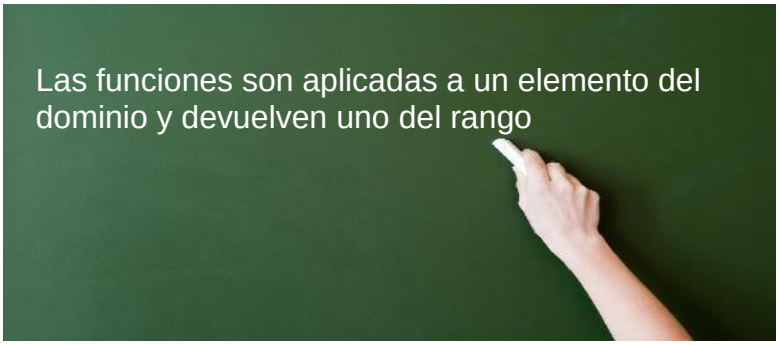
Dominio

Evaluados dentro de la función hasta el último momento y solo si se requieren para calcular el resultado final (evaluación postergada).

Rango

Son los valores que resultan.

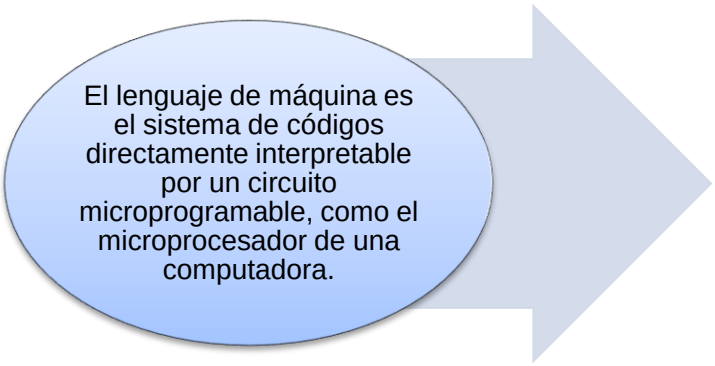
Una definición de función especifica el dominio y rango, de manera implícita o explicita, junto con una expresión que describe la correspondencia.



Las funciones matemáticas no producen efectos laterales. Dado un mismo conjunto de argumentos, una función matemática producirá siempre el mismo resultado; por ejemplo, de una función que arroje el mayor de tres números dados, siempre se va a esperar este resultado, independientemente de los valores que se ingresen.

1.3. Lenguaje máquina

Los circuitos microprogramables son sistemas digitales, lo que significa que trabajan con dos únicos niveles de tensión. Dichos niveles, por abstracción, se simbolizan con el cero (0) y el uno (1), por eso el lenguaje de máquina sólo utiliza estos signos.



Este lenguaje está compuesto por un conjunto de instrucciones que determinan acciones que serán realizadas por la máquina. Un programa de computadora consiste en una cadena de instrucciones de lenguaje de máquina (más los datos). Estas instrucciones son normalmente ejecutadas en secuencia, con eventuales cambios de flujo causados por el propio programa o eventos externos. El lenguaje de máquina es específico de cada máquina o arquitectura de la máquina, aunque el conjunto de instrucciones disponibles pueda ser similar entre ellas.

1.4. Lenguajes de bajo nivel

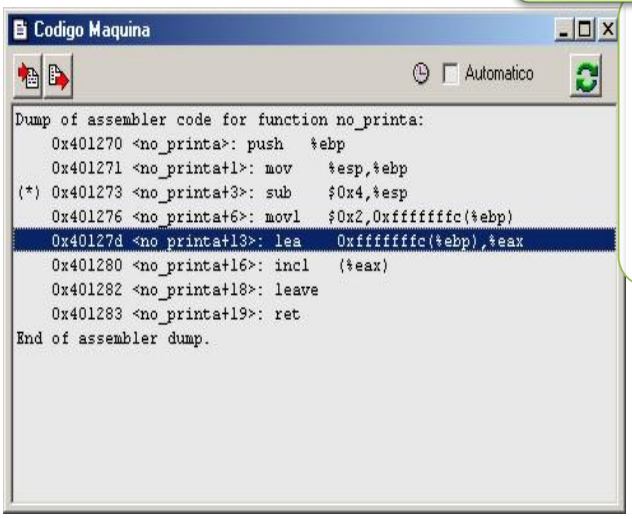
Un lenguaje de bajo nivel es fácilmente trasladado a lenguaje de máquina. La palabra 'bajo' se refiere a la reducida abstracción entre el lenguaje y el hardware.

Lenguaje de programación de bajo nivel

Es el que proporciona poca o ninguna abstracción del microprocesador de una computadora. Consecuentemente, es fácil su trasladado al lenguaje máquina.

Del lenguaje de bajo nivel sigue el lenguaje de medio nivel denominado ensamblador, que contiene instrucciones que realizan ciertas tareas básicas (ejemplos: add, jump, move, etcétera).

El término ensamblador (del inglés *assembler*)



- Se refiere a un tipo de programa informático que se encarga de traducir un archivo fuente escrito en un lenguaje ensamblador, a un archivo objeto que contiene código máquina, ejecutable directamente por la máquina para la que se ha generado.

1.5. Lenguajes de alto nivel

Los lenguajes de alto nivel se caracterizan por expresar los algoritmos de una manera sencilla y adecuada a la capacidad cognitiva humana, en lugar de la capacidad ejecutora de las máquinas.

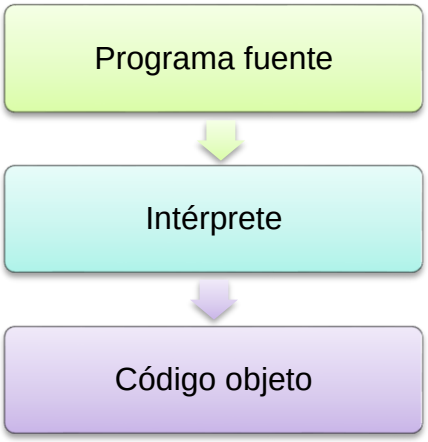
Ejemplos de lenguajes de alto nivel	
*** C++**	Es un lenguaje de programación, diseñado a mediados de los años 80, por Bjarne Stroustrup. Por otro lado, es un lenguaje que abarca dos paradigmas de la programación: la programación estructurada y la programación orientada a objetos.
*** Fortran**	Es un lenguaje de programación desarrollado en los años 50 y activamente utilizado desde entonces. Acrónimo de "Formula Translator", Fortran se utiliza principalmente en aplicaciones científicas y análisis numérico.
*** Java**	Es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. Las aplicaciones java están típicamente compiladas en un bytecode, aunque la compilación en código máquina nativo también es posible.
*** Perl**	Lenguaje práctico para la extracción e informe. Es un lenguaje de programación diseñado por Larry Wall creado en 1987. Perl toma características de C, del lenguaje interpretado shell sh, AWK, sed, Lisp y, en un grado inferior, muchos otros lenguajes de programación.
*** PHP**	Es un lenguaje de programación usado frecuentemente para la creación de contenido para sitios web, con los cuales se puede programar las páginas html y los códigos de fuente. PHP es un acrónimo que significa "PHP Hypertext Pre-processor" (inicialmente PHP Tools, o, Personal Home Page Tools), y se trata de un lenguaje interpretado usado para la creación de aplicaciones para servidores, o creación de contenido dinámico para sitios web. Últimamente se usa también para la creación de otro tipo de programas incluyendo aplicaciones con interfaz gráfica usando las librerías Qt o GTK+.

*** Python**	Es un lenguaje de programación creado por Guido van Rossum en el año 1990. En la actualidad Python se desarrolla como un proyecto de código abierto, administrado por la Python Software Foundation. La versión estable del lenguaje es de septiembre de 2006 y la versión 3.2.1 del 22 de marzo 2011.

1.6. Intérpretes

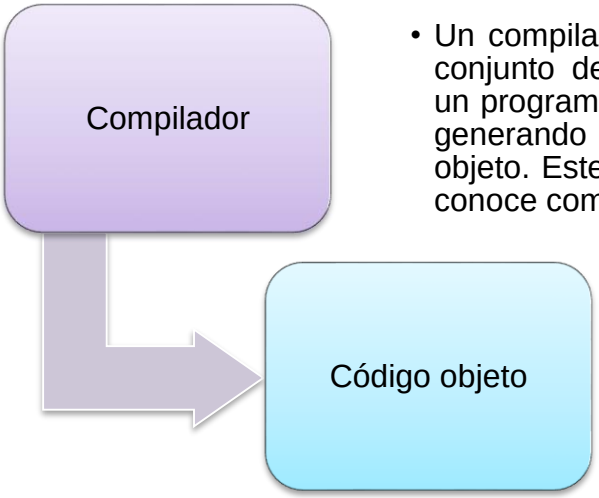
Un intérprete es un programa que analiza y ejecuta un código fuente, toma un código, lo traduce y a continuación lo ejecuta; y así sucesivamente lo hace hasta llegar a la última instrucción del programa, siempre y cuando no se produzca un error en el proceso.

Como ejemplo de lenguajes interpretados tenemos a: PHP, Perl y Python, por mencionar algunos.



Funcionamiento de un intérprete

1.7. Compiladores



- Un compilador es un programa (o un conjunto de programas) que traduce un programa escrito en código fuente, generando un programa en código objeto. Este proceso de traducción se conoce como compilación.

- Posterior a esto, al código objeto se le agregan las librerías a través de un programa llamado *linker*, y se obtiene el código ejecutable.

Como ejemplo de lenguajes que utilizan un compilador tenemos a C, C++, Visual Basic.

En C, el compilador lee el programa y lo convierte a código objeto. Una vez compilado, las líneas de código fuente dejan de tener sentido. Este código objeto puede ser ejecutado por la computadora. El compilador de C incorpora una biblioteca estándar que proporciona las funciones necesarias para llevar a cabo las tareas más usuales.

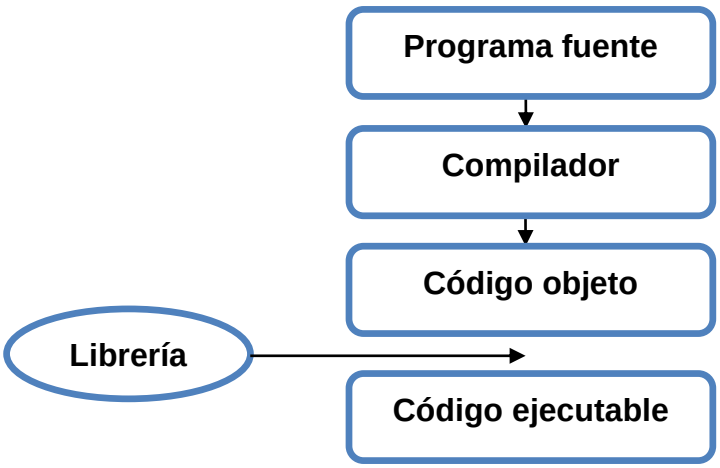
1.8. Fases de la compilación

La compilación permite crear un programa de computadora que puede ser ejecutado por una computadora.

La compilación de un programa se hace en tres pasos.

1. Creación del código fuente,
2. Compilación del programa y
3. Enlace del programa con las funciones necesarias de la biblioteca.

La forma en que se lleve a cabo el enlace variará entre distintos compiladores, pero la forma general es:



Proceso de Compilación

1.9. Notación BNF

La notación BNF o *Backus-Naur Form* es una sintaxis que se utiliza para describir a las gramáticas libres de contexto o lenguajes formales:

<símbolo> :: = <expresión con símbolos>

En la cual, *símbolo* es un *no* terminal y *expresión* es una secuencia de símbolos terminales. Observe el siguiente ejemplo:

<datos trabajador>::= <nombre del trabajador>“,”<edad>“,”<sexo>“,” <salario diario>

Los datos del trabajador consisten en el nombre, seguida por una coma, seguida por la edad, seguida por una coma, seguida por el sexo, seguida por una coma y seguida por el salario diario.

1.10. Sintaxis, léxico, semántica

Sintaxis

Se refiere al conjunto de reglas sintácticas que indican el orden y la disposición correcta de los símbolos del alfabeto del lenguaje de programación y que representa a la gramática del lenguaje; esta gramática es la que define formalmente la sintaxis del propio lenguaje de programación.

Normalmente, la estructura sintáctica se representa mediante los diagramas con la notación BNF (*Backus-Naur Form*), que ayudan a entender la producción de los elementos sintácticos.

Léxico

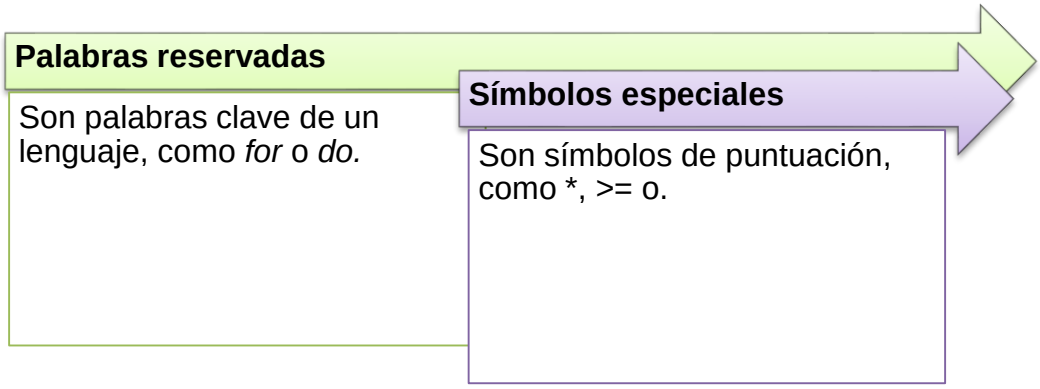
La estructura léxica de un lenguaje de programación está conformada por la estructura de sus *tokens*, los cuáles pueden ser:

Identificadores

Son palabras con un significado definido en el lenguaje de programación.

Constantes o literales

Una palabra, letra o número puede ser una constante.



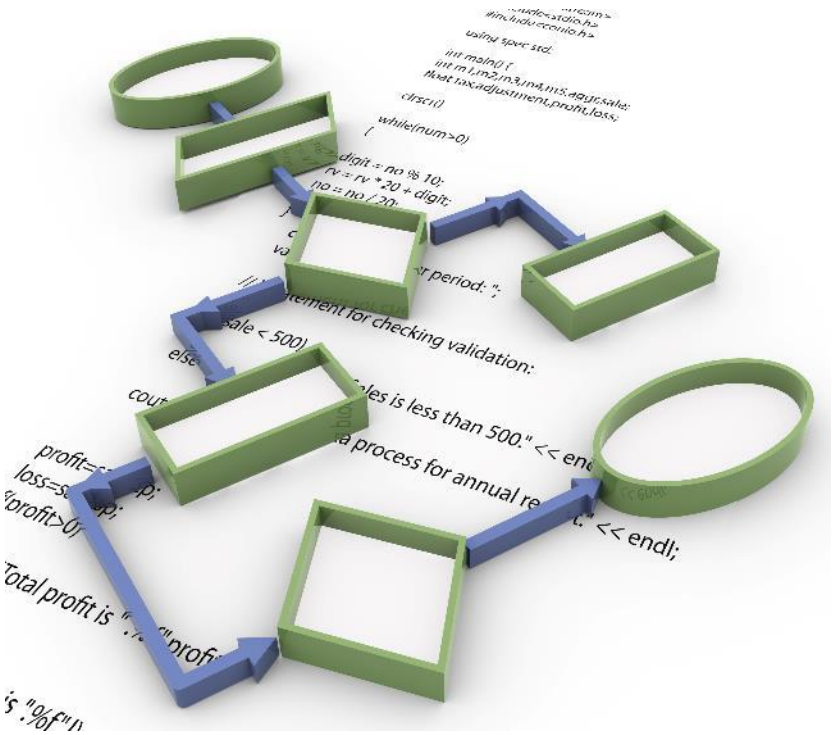
Una vez definida la estructura de los *tokens*, se define su estructura sintáctica.

Semántica

La semántica se refiere al significado y funcionamiento de un programa, normalmente el algoritmo de un programa se puede representar mediante un diagrama de flujo o un pseudocódigo, una vez solucionada la lógica del programa, se procede a traducirlo a la sintaxis de un lenguaje de programación específico.

RESUMEN

En esta unidad se desarrollaron los conceptos básicos de la programación, entendida como la implementación de un algoritmo (serie de pasos para resolver un problema) en un lenguaje de programación, dando como resultado un programa ejecutable. Se trataron diversos temas relacionados con la programación, como la programación estructurada, además del funcionamiento de intérpretes y compiladores.



BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Cairó (2003)	2	250-252
Joyanes (2003)	2	47-74
Kenneth (2004)	1	1-30
	4	69-112

UNIDAD 2.

Tipos de datos elementales

(variables, constantes, declaraciones, expresiones y estructura de un programa)

UNAM FCA
CEDIGEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Deberá conocer los componentes básicos de la programación y la estructura de un programa.

TEMARIO DETALLADO

(6 horas)

2. Tipos de datos elementales. (Variables, constantes, declaraciones, expresiones y estructura de un programa)

2.1. Tipos de datos
2.2. Palabras reservadas
2.3. Identificadores
2.4. Operadores
2.5. Expresiones y reglas de prioridad
2.6. Variables y constantes
2.7. Estructura de un programa

INTRODUCCIÓN

Un tipo de dato lo podemos definir a partir de sus valores permitidos (entero, carácter, etc.); por otro lado, las palabras reservadas se utilizan en un lenguaje de programación para fines específicos y no pueden ser utilizadas por el programador; los identificadores se utilizan para diferenciar variables y constantes en un programa. Las variables permiten que un identificador pueda tomar varios valores, por el contrario, las constantes son valores definidos que no pueden cambiar durante la ejecución del programa. Las expresiones se forman combinando constantes, variables y operadores, todos estos elementos permiten la creación de un programa informático.

2.1. Tipos de datos

Tipos de datos elementales

Las formas de organizar datos están determinadas por los tipos de datos definidos en el lenguaje.

Un tipo de dato determina el rango de valores que puede tomar el objeto, las operaciones a que puede ser sometido y el formato de almacenamiento en memoria.

En el lenguaje C:

a) Existen tipos predefinidos.

b) El usuario puede definir otros tipos, a partir de los básicos.

Esta tabla describe los tipos de datos que se pueden utilizar en el lenguaje C.

Tipos de datos elementales

TIPO	RANGO DE VALORES	TAMAÑO EN BYTES	DESCRIPCIÓN
char	-128 a 127	1	Para una letra o un dígito.
unsigned char	0 a 255	1	Letra o número positivo.
Int	-32.768 a 32.767	2	Para números enteros.
unsigned int	0 a 65.535	2	Para números enteros.
long int	±2.147.483.647	4	Para números enteros
unsigned long int	0 a 4.294.967.295	4	Para números enteros

float	3.4E-38 decimales(6)	6	Para números con decimales
double	1.7E-308 decimales(10)	8	Para números con decimales
long double	3.4E-4932 decimales(10)	10	Para números con decimales

Ejemplo de utilización de tipos de datos en C.

Ejemplo de utilización de tipos de datos en C.

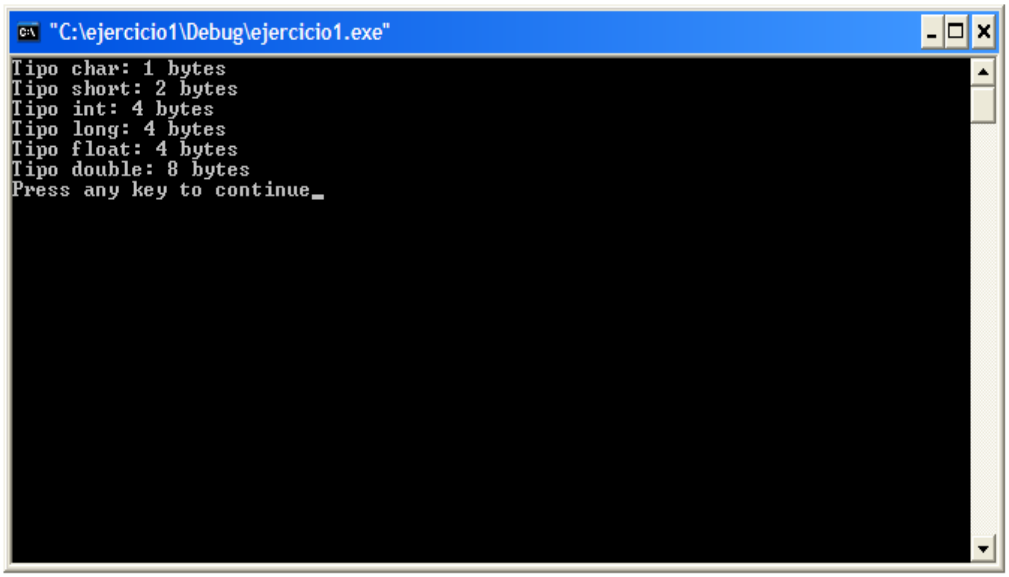
```
int numero;

int numero = 10;

float numero = 10.23;
```

Veamos un programa que determina el tamaño en bytes de los tipos de datos fundamentales en C.

Ejercicio 1.1

```
# include <stdio.h>
void main()
{
char c;
short s;
int I;
long l;
float f;
double d;
printf ("Tipo char: %d bytes\n", sizeof(c));
printf ("Tipo short: %d bytes\n", sizeof(s));
printf ("Tipo int: %d bytes\n", sizeof(i));
printf ("Tipo long: %d bytes\n", sizeof(l));
printf ("Tipo float: %d bytes\n", sizeof(f));
printf ("Tipo double: %d bytes\n", sizeof(d));
}
```

El resultado de este programa es el siguiente:



Tipos definidos por el usuario

C permite nuevos nombres para tipos de datos. Realmente no se crea un nuevo tipo de dato, sino que se define uno nuevo para un tipo existente. Esto ayuda a hacer más transportables los programas que dependen de las máquinas, sólo habrá que cambiar las sentencias *typedef* cuando se compile en un nuevo entorno. Las sentencias *typedef* permiten la creación de nuevos tipos de datos.

Sintaxis:

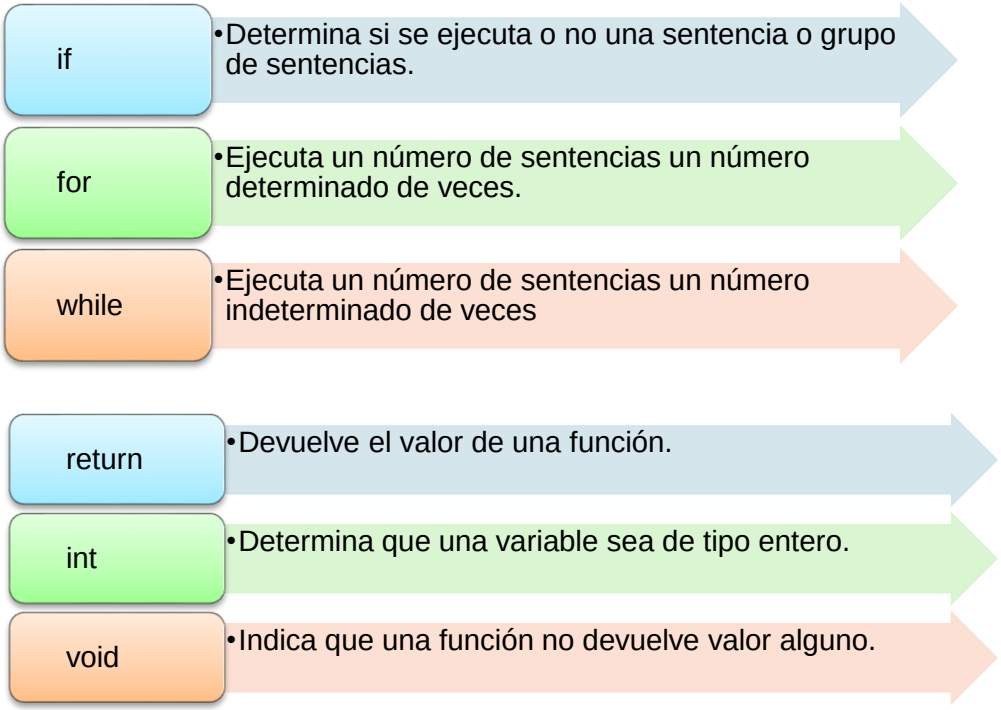
```
typedef tipo nuevo_nombre;
nuevo_nombre nombre_variable[=valor];
```

Por ejemplo, se puede crear un nuevo nombre para *float* usando:

```
typedef float balance;
```

2.2. Palabras reservadas

Las palabras reservadas son símbolos cuyo significado está predefinido y no se pueden usar para otro fin; aquí tenemos algunas palabras reservadas en el lenguaje C.



Observa el siguiente ejemplo de un programa que obtiene la suma de los números pares del 1 al 10 en este se ocupan varias de las palabras reservadas mencionadas anteriormente:

```
#include <stdio.h>
main(void)
{  int x, suma=0; //declaración de variables enteras “x” y “suma”
   for (x=1; x<=10; x++) //estructura de repetición for
   {  while (x%2==0) // estructura iterativa while
      suma=suma+x; //la variable suma acumula la sumatoria de los pares
   }
   //la siguiente instrucción muestra en pantalla la suma de los pares
   printf(“La suma de los números pares del 1 al 10 es: %i”,suma);
   return 0;
}
```

2.3. Identificadores

- Son secuencias de carácter que se pueden formar usando letras, cifras y el carácter de subrayado “_”
- Se usan para dar nombre a los objetos que se manejan en un programa: tipos, variables, funcionales.
- Deben comenzar por letra o por “_”
- Se distinguen entre mayúsculas y minúsculas.
- Se deben definir en sentencias de declaración antes de ser usados.

Cabe destacar que los identificadores pueden contener cualquier número de caracteres, pero solamente los primeros 32 son significativos para el compilador.

Sintaxis:

```
int i, j, k;
float largo, ancho, alto;
enum colores {rojo, azul, verde}
color1, color2;
```

Un ejemplo de un identificador no válido sería:	Int 8identifica.

2.4. Operadores

C es un lenguaje muy rico en operadores incorporados, es decir, implementados al realizarse el compilador. Se pueden utilizar operadores: aritméticos, relacionales y lógicos. También se definen operadores para realizar determinadas tareas, como las asignaciones.

Asignación

Los operadores de asignación son aquellos que nos permiten modificar el valor de una variable, el operador de asignación básico es el “igual a” (=), que da el valor que lo sigue a la variable que lo precede.

Al utilizarlo se realiza esta acción: el operador destino (parte izquierda) debe ser siempre una variable, mientras que en la parte derecha puede estar cualquier expresión válida. Con esto el valor de la parte derecha se asigna a la variable de la derecha.

Sintaxis

variable=valor;
variable=variable1;
variable=variable1=variable2=variableN=valor;

Operadores aritméticos

Los operadores aritméticos pueden aplicarse a todo tipo de expresiones. Son utilizados para realizar operaciones matemáticas sencillas, aunque uniéndolos se puede realizar cualquier tipo de operaciones.
En la siguiente tabla se muestran todos los operadores aritméticos.

Operador	Descripción	Ejemplo
-	Resta	a-b
+	Suma	a+b
*	Multiplica	a*b
/	Divide	a/b
%	Módulo (resto de una división)	a%b
-	Signo negativo	-a
--	Decremento en 1.	a--
++	Incrementa en 1.	b++

a) Corresponden a las operaciones matemáticas de suma, resta, multiplicación, división y módulo.
b) Son binarios porque cada uno tiene dos operandos.
c) Hay un operador unario menos "-", pero no hay operador unario más "+":

-3 es una expresión correcta +3 no es una expresión correcta
La división de enteros devuelve el cociente entero y desecha la fracción restante: 1/2 tiene el valor 0 3/2 tiene el valor 1 -7/3 tiene el valor -2
El operador módulo se aplica así: con dos enteros positivos, devuelve el resto de la división. 12%3 tiene el valor 0 12%5 tiene el valor 2
Los operadores de incremento y decremento son unarios. a) Tienen la misma prioridad que el menos "-" unario. b) Se asocian de derecha a izquierda. c) Pueden aplicarse a variables, pero no a constantes ni a expresiones. d) Se pueden presentar como prefijo o como sufijo. e) Aplicados a variables enteras, su efecto es incrementar o decrementar el valor de la variable en una unidad: ++i Es equivalente a i=i+1; --i Es equivalente a i=i-1;
Cuando se usan en una expresión, se produce un efecto secundario sobre la variable: ♦ El valor de la variable se incrementa antes o después de ser usado. ♦ Con ++a el valor de "a" se incrementa antes de evaluar la expresión. ♦ Con a++ el valor de "a" se incrementa después de evaluar la expresión. ♦ Con a el valor de "a" no se modifica antes ni después de evaluar la expresión.

Ejemplos:

a=2*(++c), se incrementa el valor de "c" y se evalúa la expresión después.

Es equivalente a: c=c+1; a=2*c;

A [++i]=0 se incrementa el valor de "i" y se realiza la asignación después.
Es equivalente a: i=i+1; a[i]=0;

A [i++], se realiza la asignación con el valor actual de "i", y se incrementa el valor de "i" después.

Es equivalente a: a[i]=0; i=i+1;

Lógicos y relacionales

Los operadores relacionales hacen referencia a la relación entre unos valores y otros; los lógicos, a la forma en que esas relaciones pueden conectarse entre sí. Los veremos a la par por la estrecha relación en la que trabajan.

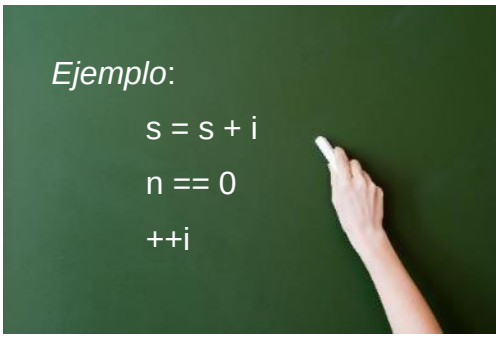
Operadores relacionales	
Operador	**Descripción**
<	Menor que.
>	Mayor que.
<=	Menor o igual.
>=	Mayor o igual
= =	Igual
¡! =	Distinto

Operadores lógicos	
Operador	**Descripción**
&&	Y (AND) ||	O (OR)
¡!	NO (NOT)

Para que quede más clara la información, observa los siguientes ejercicios. (**ANEXO 2**).

2.5. Expresiones y reglas de prioridad

Una expresión se forma combinando constantes, variables, operadores y llamadas a funciones.



Una expresión representa un valor, el resultado de realizar las operaciones indicadas siguiendo las reglas de evaluación establecidas en el lenguaje.

Con expresiones se forman sentencias; con éstas, funciones, y con éstas últimas se construye un programa completo.

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas.

Una expresión consta de operadores y operandos. Según sea el tipo de datos que manipulan, se clasifican las expresiones en:

a) Aritméticas.

- Contienen a los operadores aritméticos *, /, %, +, y -. En este orden se llevan a cabo las operaciones, a menos que se utilicen paréntesis para cambiar esta prioridad.

b) Relacionales.

- Este tipo de expresiones utilizan operadores relacionales para comparar operandos. El resultado de la comparación es un valor de tipo booleano: verdadero o falso.

c) Lógicas.

- Son expresiones que actúan sobre dos operandos. Los operadores son los siguientes:

  La conjunción (&&) obtiene un valor verdadero únicamente si los dos operandos son verdaderos, de lo contrario, el resultado es falso.

  En la disyunción (||) se obtiene el resultado verdadero si ambos operandos o alguno de estos son verdaderos, el resultado es falso si ambos operadores son falsos.

  En la negación (!) si la expresión es verdadera el resultado es falso, viceversa, si la expresión es falsa el resultado es verdadero.

Prioridades de los operadores aritméticos

Son prioridades de los operadores matemáticos:

Los operadores en una misma expresión con igual nivel de prioridad se evalúan de izquierda a derecha.

1. Todas las expresiones entre paréntesis se evalúan primero. Las expresiones con paréntesis anidados se evalúan de adentro hacia fuera, el paréntesis más interno se evalúa primero.

2. Dentro de una misma expresión los operadores se evalúan en el siguiente orden:
( ) Paréntesis
^ Exponenciación
*, /, mod Multiplicación, división, módulo. (El módulo o mod es el resto de una división)
+, - Suma y resta.

Ejemplos:

a.	4 + 2 * 5 = 14	Primero se multiplica y después se suma
b.	23 * 2 / 5 = 9.2	46 / 5 = 9.2
c.	3 + 5 * (10 - (2 + 4)) = 23	3 + 5 * (10 - 6) = 3 + 5 * 4 = 3 + 20 = 23
d.	2.1 * (1.5 + 3.0 * 4.1) = 28.98	2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98

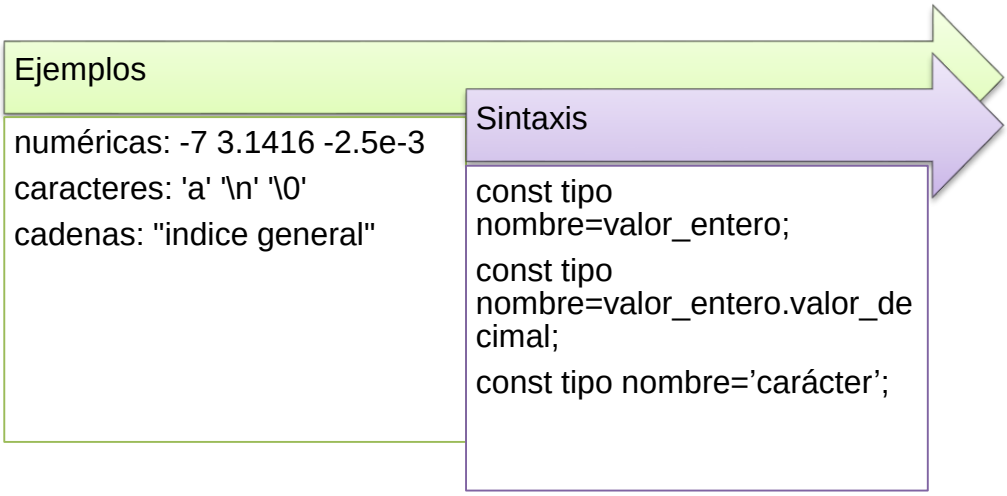
2.6. Variables y constantes

Constantes

Las constantes se refieren a los valores fijos que no pueden ser modificados por el programa. Las constantes de carácter van encerradas en comillas simples. Las constantes enteras se especifican con números sin parte decimal, y las de coma flotante, con su parte entera separada por un punto de su parte decimal.

Las constantes son entidades cuyo valor no se modifica durante la ejecución del programa.

Hay constantes de varios tipos:



Otra manera de usar constantes es a través de la directiva #define. Éste es un identificador y una secuencia de caracteres que se sustituirá cada vez que se encuentre éste en el archivo fuente. También pueden ser utilizados para definir valores numéricos constantes.

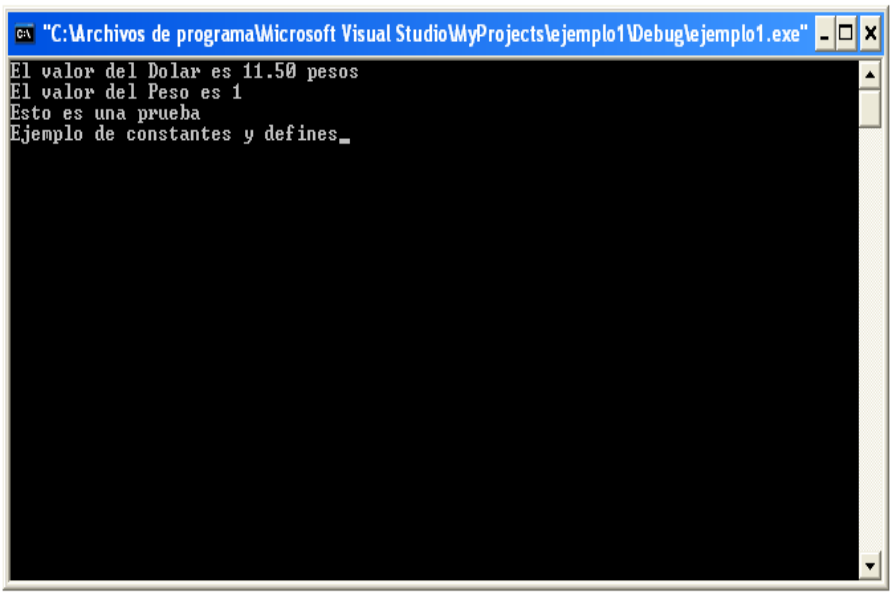
SINTAXIS:

```
#define IDENTIFICADOR valor_numerico
#define IDENTIFICADOR “cadena”
```

Ejercicio

```
#include <stdio.h>
#include <conio.h>
#define DOLAR 11.50
#define TEXTO "Esto es una prueba"
const int peso=1;
void main(void)
{
printf("El valor del Dólar es %.2f pesos",DOLAR);
        printf("\nEl valor del Peso es %d ",peso);
printf("\n%s",TEXTO);
printf("\nEjemplo de constantes y defines");
        getch(); /*función que detiene la ejecución del programa hasta que
el usuario pulse alguna tecla, viene incluida en la librería conio.h*/
}
```

El resultado del programa es el siguiente:



Variables

Unidad básica de almacenamiento de valores. La creación de una variable es la *combinación* de un *identificador,* un *tipo* y un *ámbito*. Todas las variables en C deben ser declaradas antes de ser usadas.

Las variables, también conocidas como *identificadores*, deben cumplir las siguientes reglas:

La longitud puede ir de 1 carácter a 31.

El primero de ellos debe ser siempre una letra.

No puede contener espacios en blanco, ni acentos y caracteres gramaticales.

Hay que tener en cuenta que el compilador distingue entre mayúsculas y minúsculas.

SINTAXIS:

```
tipo nombre=valor_numerico;
tipo nombre='letra';
tipo nombre[tamaño]="cadena de letras",
tipo nombre=valor_entero.valor_decimal;
```

Conversión

Las conversiones (*casting*) automáticas pueden ser controladas por el programador. Bastará con anteponer y encerrar entre paréntesis el tipo al que se desea convertir.

Este tipo de conversiones sólo es temporal y la variable por convertir mantiene su valor.

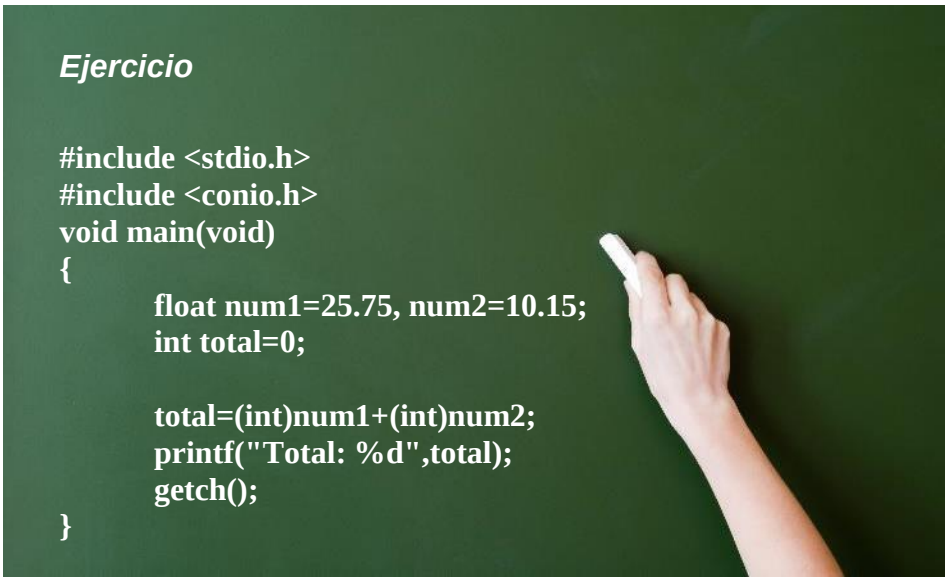
SINTAXIS:

```
variable_destino=(tipo)variable_a_convertir;
variable_destino=(tipo)(variable1+variable2+variableN);
```

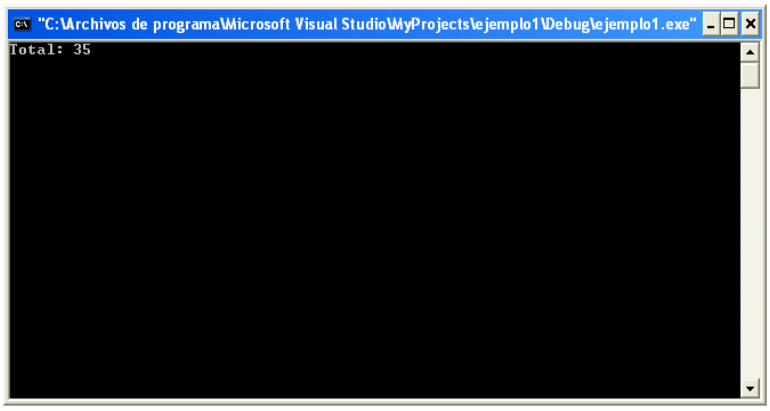
Ejemplo:

Convertimos 2 variables *float* para guardar la suma en un entero.

La biblioteca *conio.h* define varias funciones utilizadas en las llamadas a rutinas de entrada/salida por consola de dos.



El resultado del programa es el siguiente:



Ámbito de variables

Según el lugar donde se declaren las variables tendrán un ámbito. Según el ámbito de las variables pueden ser utilizadas desde cualquier parte del programa o únicamente en la función donde han sido declaradas.

Por ejemplo: En el siguiente programa se utilizan variables que pueden utilizarse en todo el programa (variables *globales*) y variables que solo se pueden utilizar dentro de la función (variables *locales*); el objetivo de este programa es intercambiar los valores almacenados en las variables num1 y num2:

```
#include <stdio.h>
int num1=5,num2=10; //variables que pueden emplearse en
todo el programa
void cambio(int &num1,int &num2); //prototipo de la
función cambio
void main (void)
{
printf("Intercambio de valores: ");
cambio(num1,num2); //llamada a función cambio pasando
dos argumentos
}
void cambio(int &num1,int &num2)
{
int aux; //variable auxiliar cuyo ámbito es únicamente
dentro de la función cambio
//las siguientes tres líneas se intercambian valores
mediante la variable aux
aux=num1;
num1=num2;
num2=aux;
}
```

Tipos de variables

Locales	•Cuando se declaran dentro de una función. Pueden ser referenciadas (utilizadas) por sentencias que estén dentro de la función que han sido declaradas. No son conocidas fuera de su función. Pierden su valor cuando se sale de la función.
Globales	•Son conocidas a lo largo de todo el programa y se pueden usar desde cualquier parte del código. Mantienen sus valores durante toda la ejecución. Deben ser declaradas fuera de todas las funciones incluida main(). La sintaxis de creación no cambia nada con respecto a las variables locales.
De registro	•Otra posibilidad es que, en vez de ser mantenidas en posiciones de memoria de la computadora, se las guarde en registros internos del microprocesador. De esta manera, el acceso a ellas es más directo y rápido. Para indicar al compilador que es una variable de registro, hay que añadir a la declaración la palabra register delante del tipo. Solo se puede utilizar para variables locales.
Estáticas	•Las variables locales nacen y mueren con cada llamada y finalización de una función. Sería útil que mantuvieran su valor entre una llamada y otra sin por ello perder su ámbito. Para conseguir eso se añade a una variable local la palabra static delante del tipo.
Externas	•Debido a que en C es normal la compilación por separado de pequeños módulos que componen un programa completo, puede darse el caso de que deba utilizar una variable global que se conozca en los módulos que nos interesen sin perder su valor. Añadiendo delante del tipo la palabra extern y definiéndola en los otros módulos como global ya tendremos nuestra variable global.

2.7. Estructura de un programa

Ya hemos visto varios programas en C, sin embargo, no hemos revisado su estructura, por lo que analizaremos sus partes, con el último programa visto:

```
#include <stdio.h>          <-BIBLIOTECA
#include <conio.h>          <-BIBLIOTECA
```

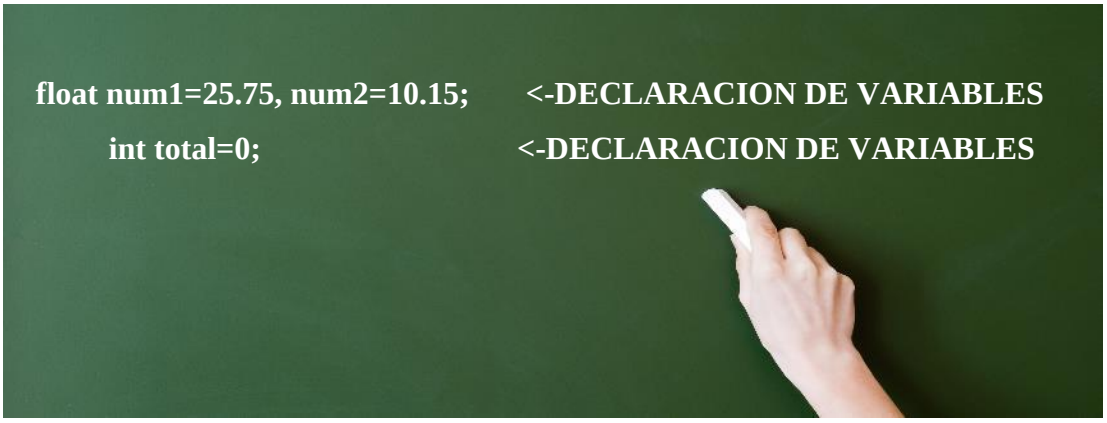
Las bibliotecas son repositorios de funciones, la biblioteca stdio.h contiene funciones para la entrada y salida de datos, un ejemplo de esto es la función printf.

```
void main(void)   <-FUNCION PRINCIPAL
```

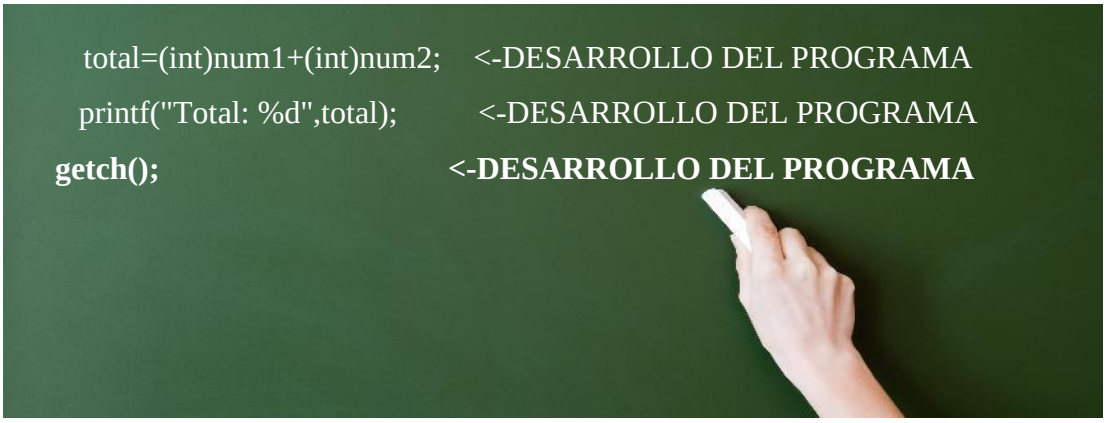
La función main es la función principal de todo programa en C.

```
{    LLAVES DE INICIO
```

Las llaves de inicio y de fin, indican cuando inicia y termina un programa.



En esta parte se declaran las variables que se van usar, su tipo, y en su caso se inicializan.



En esta parte, como su nombre lo indica, se desarrolla el programa.

Estas son las partes más importantes de un programa en C.

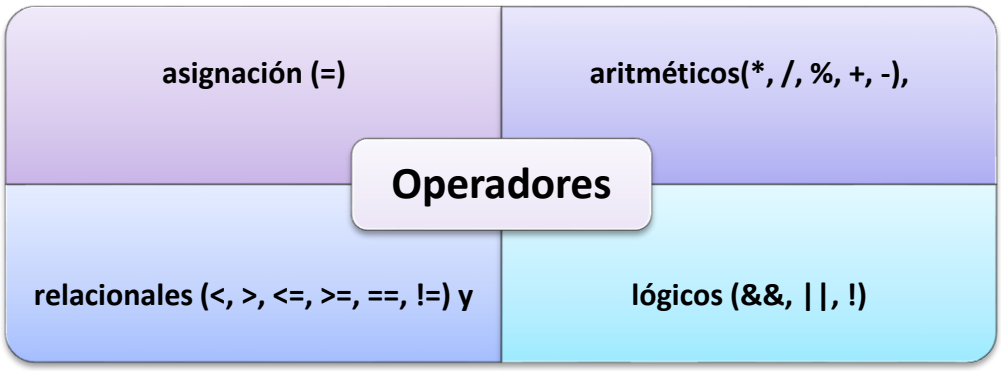
RESUMEN

En esta unidad se vieron los conceptos de variable, constante, expresión y palabra reservada, así como ejemplos de su utilización en lenguajes de programación, tomando como ejemplo el lenguaje C.

Los tipos de datos se definen a partir de tipos definidos (entero, carácter, etc.); las palabras reservadas (símbolos con significado predefinido, p.e.: if, for, while, etcétera) se utilizan para fines específicos y no los puede utilizar el programador; los identificadores (secuencia de caracteres para nombrar objetos) se utilizan para definir variables (valores que cambian constantemente) y constantes (valores que no cambian); las expresiones combinan las constantes, variables y operadores.

Un tipo de dato determina el rango de valores que puede tomar éste, las operaciones y el formato de su almacenamiento en memoria. Hay tipos predefinidos (char, int, float, double, entre otros) y tipos nuevos definidos por el usuario creados con *typedef*. Cada tipo de dato tiene un tamaño en bytes.

El lenguaje C cuenta con operadores de asignación (=), aritméticos (*, /, %, +, -), relacionales (<, >, <=, >=, ==, !=) y lógicos (&&, ||, !) que pueden aplicarse sobre todo tipo de expresiones. Las expresiones que tienen paréntesis se evalúan primero desde el paréntesis más interno hacia fuera.

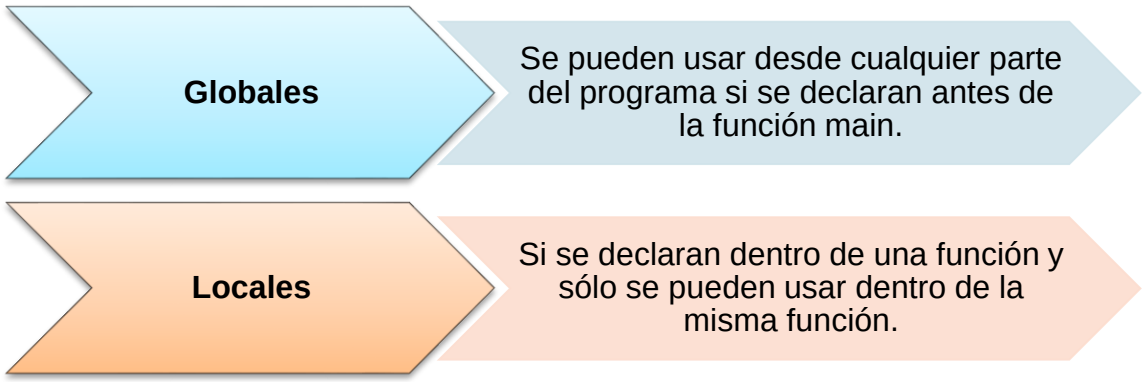


Los operadores se evalúan en el siguiente orden: (), ^, *, /, %, +, -. Los operadores, en una misma expresión con el mismo nivel de prioridad, se evalúan de izquierda a derecha.

Las variables se pueden convertir (casting) a un tipo de datos diferente, anteponiendo a la variable un paréntesis con el tipo de dato al que desees convertirla, p.ej.:

```
variable_destino=(tipo de dato) variable_a_convertir.
```

Las variables tienen un ámbito de acuerdo al lugar donde se declaren:



También hay variables:

Registro	Estáticas	Externas
• Almacenadas en los registros internos del microprocesador.	• Variables locales que mantienen su valor entre las llamadas a funcione.	• Variable global que es conocida por todos los módulos que conforman el programa.

La estructura de un programa está compuesta de bibliotecas (#include), función principal (main), llaves ({}), y dentro de éstas, la declaración de variables y desarrollo del programa (conjunto de instrucciones).

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Cairó (2003)	2	250-252
Joyanes (2003)	2	47-110
	3	113-144
	4	151-172
	5	177-209

UNIDAD 3

Control de flujo

UNAM FCA CEDIGEC Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Podrá utilizar las principales estructuras de la programación.

TEMARIO DETALLADO

(14 horas)

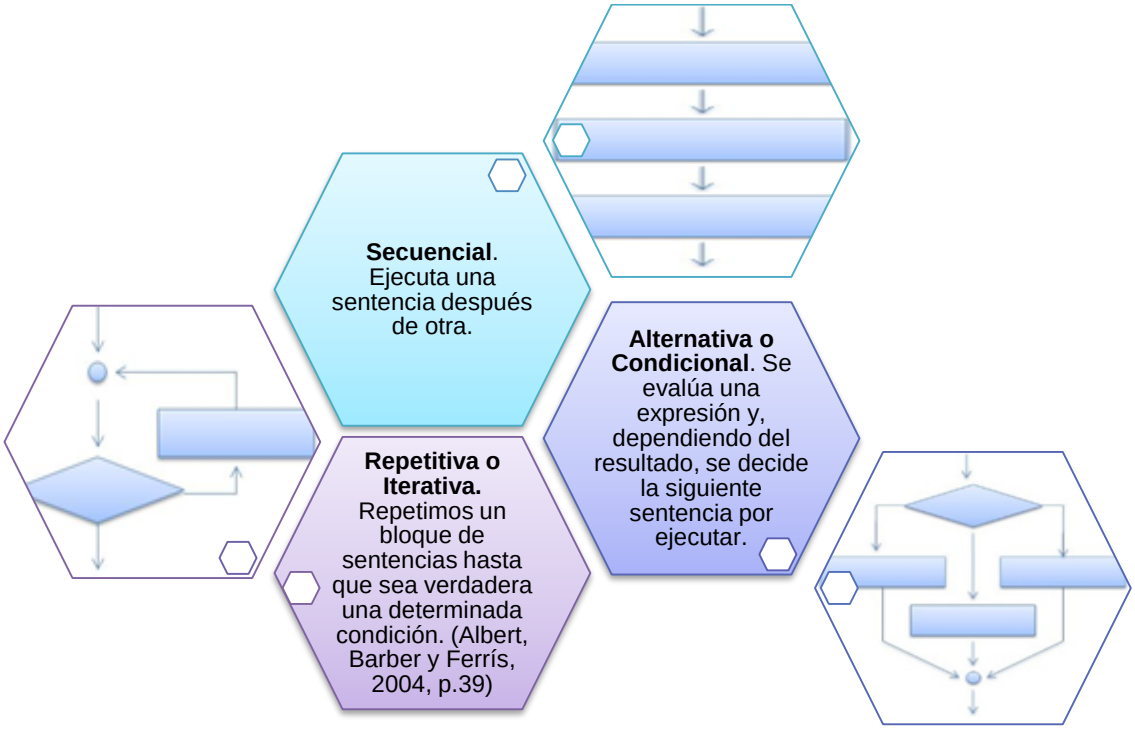
3. Control de flujo

- 3.1. Estructura secuencial
- 3.2. Estructura alternativa
- 3.3. Estructura repetitiva

INTRODUCCIÓN

A finales de los años 60, surgió una nueva forma de programar que daba lugar a programas fiables y eficientes, además de que estaban escritos de tal manera que facilitaba su comprensión.

El teorema del programa estructurado, demostrado por Böhm-Jacopini, demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:



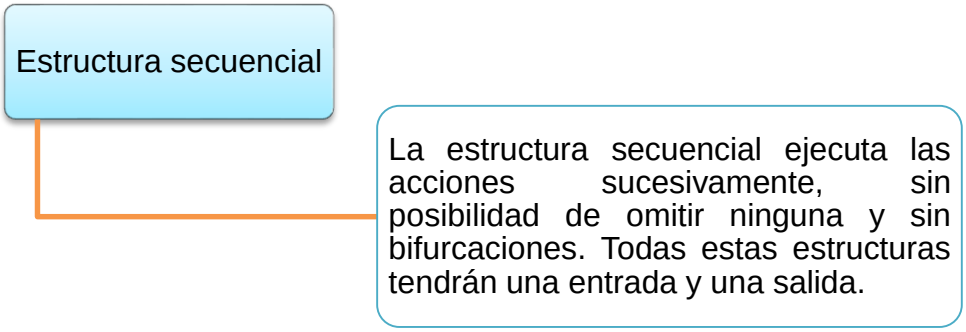
Solamente con estas tres estructuras se pueden escribir cualquier tipo de programa.

La programación estructurada crea programas claros y fáciles de entender, además los bloques de código son auto explicativos, lo que facilita la documentación. No obstante,

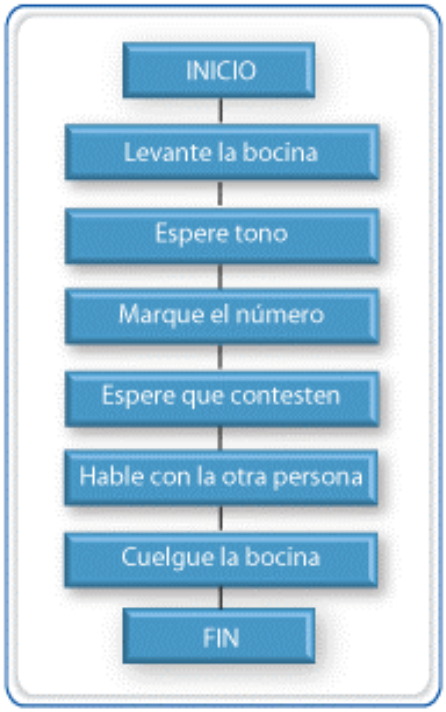
cabe destacar que en la medida que los programas aumentan en tamaño y complejidad, su mantenimiento también se va haciendo difícil.

3.1. Estructura secuencial

El control de flujo se refiere al orden en que se ejecutan las sentencias del programa. A menos que se especifique expresamente, el flujo normal de control de todos los programas es secuencial.



Ejemplo de un diagrama de flujo

Inicio Acción a_1 Acción a_2 . . Acción a_n Fin	

Veamos algunos programas que utilizan la estructura secuencial, empleando el lenguaje C.

Ejemplo

Supón que un individuo desea invertir su capital en un banco y desea saber cuánto dinero ganará después de un mes si el banco paga a razón de 2% mensual.

```
#include <stdio.h>
void main()
{
        double cap_inv=0.0,gan=0.0;
        printf("Introduce el capital invertido\n");
        scanf("%lf",&cap_inv);
```

```
        gan=(cap_inv * 0.2);
        printf("La ganancia es: %lf\n",gan);
}
```

Como podrá notarse, no hay posibilidad de omitir alguna sentencia del anterior programa, todas las sentencias serán ejecutadas.

Ve más ejemplos **de estructura secuencial** en los siguientes ejercicios.

Ejercicio 1

Un vendedor recibe un sueldo base más un 10% extra por comisión de sus ventas. El vendedor desea saber cuánto dinero obtendrá por concepto de comisiones por las tres ventas que realiza en el mes y el total que recibirá, tomando en cuenta su sueldo base y comisiones.

```
#include <stdio.h>
void main()
{
	double tot_vta,com,tpag,sb,v1,v2,v3;
	printf("Introduce el sueldo base:\n");
	scanf("%lf",&sb);
	printf("Introduce la venta 1:\n");
	scanf("%lf",&v1);
	printf("Introduce la venta 2:\n");
	scanf("%lf",&v2);
	printf("Introduce la venta 3:\n");
	scanf("%lf",&v3);
	tot_vta = (v1+v2+v3);
	com = (tot_vta * 0.10);
	tpag = (sb + com);
	printf("total a pagar: %f\n",tpag);
}
```

Ejercicio 2

Una tienda ofrece un descuento de 15% sobre el total de la compra y un cliente desea saber cuánto deberá pagar finalmente por el total de las compras.

```
#include <stdio.h>
void main()
{
	double tc,d,tp;
	printf("Introduce el total de la compra:\n");
	scanf("%f",&tc);
	d = (tc*0.15);
	tp = (tc-d);
	printf("total de las compras: %f\n",tp);
}
```

Ejercicio 3

Un alumno desea saber cuál será su calificación final en la materia de Algoritmos. Dicha calificación se compone de los siguientes porcentajes:

- 55% del promedio de sus tres calificaciones parciales.
- 30% de la calificación del examen final.
- 15% de la calificación de un trabajo final.

```
#include <stdio.h>
void main()
{
        float c1,c2,c3,ef,pef,tf,ptf,prom,ppar,cf;
        printf("Introduce la calificación 1:\n");
        scanf("%f",&c1);
        printf("Introduce la calificación 2:\n");
        scanf("%f",&c2);
        printf("Introduce la calificación 3:\n");
        scanf("%f",&c3);
        printf("Introduce la calificación del examen final:\n");
        scanf("%f",&ef);
        printf("Introduce la calificación del trabajo final:\n");
        scanf("%f",&tf);
        prom = ((c1+c2+c3)/3);
        ppar = (prom*0.55);
        pef = (ef*0.30);
        ptf = (tf*0.15);
        cf = (ppar+pef+ptf);
        printf("La calificación final es: %.2f\n",cf);
}
```

3.2. Estructura alternativa

Estructura alternativa

La estructura alternativa es aquella en que la existencia o cumplimiento de la condición, implica la **ruptura** de la **secuencia** y la **ejecución** de una determinada acción. Es la manera que tiene un lenguaje de programación de provocar que el flujo de la ejecución avance y se ramifique en función de los cambios de estado de los datos.

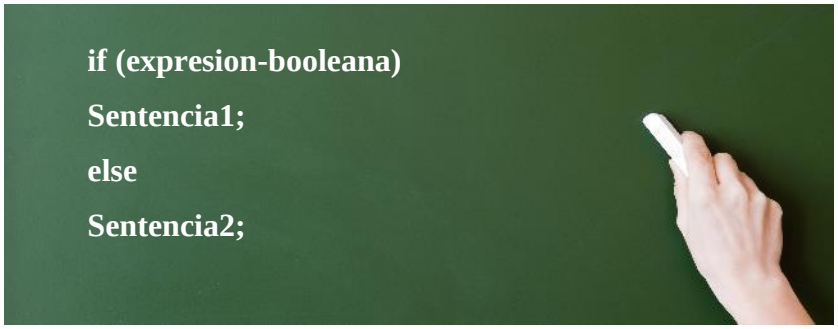
Inicio	if (expresión-booleana)
Si condición Entonces	{
Acción	Sentencia1;
De_lo_contrario	sentencia2;
Acción	}
Fin_Si	else
Fin	Sentencia3;

If-else

En la figura anterior, se muestra una **estructura alternativa simple**. La línea "*Si (condición) entonces*" corresponde a la instrucción "if (expresión-booleana)" en el lenguaje C y la línea donde indica la frase "De lo contrario" corresponde a "*else*". La ejecución de "*if*" atraviesa un conjunto de estados booleanos[1] que determinan que se ejecuten distintos fragmentos de código, es decir, si la condición evaluada en *if* es verdadera, se ejecuta la o las sentencias que están inmediatamente colocadas después

[1] Boolean: contiene valores que pueden ser sólo True o False. Las palabras clave True y False corresponden a los dos estados de las variables Boolean.

de la línea del *if* y si es falsa, se ejecutan las sentencias que se encuentran después de *else,* como se muestra a continuación:



La cláusula **else** es opcional, la expresión puede ser de cualquier tipo y más de una (siempre que se unan mediante operadores lógicos). Otra opción posible es la utilización de **if** anidados, es decir unos dentro de otros compartiendo la cláusula **else**.

Ejemplo

Realiza un programa que determine si un alumno aprueba o reprueba una materia.

```
# include <stdio.h>
main()
{
1.1.1   float examen, tareas, trabajo, final;
1.1.2   printf("Por favor introduzca la calificación de los exámenes: ");
1.1.3   scanf("%f",&examen);
1.1.4   printf("Por favor introduzca la calificación de las tareas: ");
1.1.5   scanf("%f",&tareas);
1.1.6   printf("Por favor introduzca la calificación del trabajo: ");
1.1.7   scanf("%f",&trabajo);
1.1.8   final = (examen+tareas+trabajo)/3;
1.1.9   printf("Tu calificación final es de: %.2f\n",final);
1.1.10   if(final < 6)
1.1.11   printf("Tendrás que cursar programación nuevamente\n");
1.1.12   else
1.1.13   printf("Aprobaste con la siguiente calificación: %.2f\n",final);
1.1.14   return(0); }
```

Switch

- Realiza distintas operaciones con base en el valor de la única variable o expresión.

Es una sentencia muy similar a **if-else**, pero es mucho más cómoda y fácil de comprender.

```
switch (expresión){
        case valor1:
                sentencia;
                break;
        case valor2:
                sentencia;
                break;
        case valor3:
                sentencia;
                break;
        case valorN:
                sentencia;
                break;
        default:
  }
```

```
if (expresion-booleana)
{
        Sentencia1;
        Sentencia2;
}
Else
        Sentencia3;
```

El valor de la expresión se compara con cada uno de los literales de la sentencia, **case** si coincide alguno, se ejecuta el código que le sigue, si ninguno coincide se realiza la sentencia **default** (opcional), si no hay sentencia **default** no se ejecuta nada.

La sentencia *break* realiza la salida de un bloque de código. En el caso de sentencia switch, realiza el código y cuando ejecuta *break*, sale de este bloque y sigue con la

ejecución del programa. En el caso de que varias sentencias *case* realicen la misma ejecución, se pueden agrupar utilizando una sola sentencia *break*.

```
switch (expresión){
case valor1:
case valor2:
case valor5
sentencia;
break;
case valor3:
case valor4:
sentencia;
break;
default:
}
```

Veamos un ejercicio donde se utiliza la estructura *case*. El usuario debe escoger cuatro opciones introduciendo un número, si los valores con los que se compara son números, se coloca directamente, pero si es un carácter, se debe cerrar entre comillas simples; en caso de que el usuario introduzca un número incorrecto aparecerá el mensaje “Opción incorrecta”.

Ejercicio

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
int opcion;
```

```
printf("1.ALTAS\n");
printf("2.BAJAS\n");
printf("3.MODIFICA\n");
printf("4.SALIR\n");
printf("Elegir opción: ");
scanf("%d",&opcion);
switch(opcion)
{
case 1:
printf("Opción Uno");
break;
case 2:
printf("Opción Dos");
break;
case 3:
printf("Opción Tres");
break;
case 4:
printf("Opción Salir");
break;
default:
printf("Opción incorrecta");
}
getch();
}
```

Observa más **ejemplos de estructura alternativa** en los siguientes ejercicios:

Ejercicio 1

Una persona desea saber cuánto dinero se genera por concepto de intereses sobre la cantidad que tiene en inversión en el banco. Ella decidirá reinvertir los intereses siempre y cuando sean iguales o mayores que $7,000.00. Finalmente, desea saber cuánto dinero tendrá en su cuenta.

```
#include <stdio.h>
void main()
{
  float p_int,cap,inte,capf;
  printf("Introduce el porcentaje de intereses:\n");
  scanf("%f",&p_int);
  printf("Introduce el capital:\n");
  scanf("%f",&cap);
   inte = (cap*p_int)/100;
  if (inte >= 7000)
    {
     capf=(cap+inte);
      printf("El capital final es: %.2f\n",capf);
    }
    else
     printf("El interes: %.2f es menor a 7000\n",inte);
}
```

Ejercicio 2

Determina si un alumno aprueba a reprueba un curso, sabiendo que aprobará si su promedio de tres exámenes parciales es mayor o igual que 70 puntos; y entregó un trabajo final. (No se toma en cuenta la calificación del trabajo final, sólo se toma en cuenta si lo entregó).

```
# include <stdio.h>
main()
{
  float examen1, examen2, examen3,final;
  int tra_fin=0;
  printf("Por favor introduzca la calificacion del primer examen: ");
  scanf("%f",&examen1);
  printf("Por favor introduzca la calificacion del segundo examen: ");
  scanf("%f",&examen2);
  printf("Por favor introduzca la calificacion del tercer examen: ");
  scanf("%f",&examen3);
  printf("Introduzca 1 si entrego 0 si no entrego trabajo final: ");
  scanf("%d",&tra_fin);
  final = (examen1+examen2+examen3)/3;
  if(final < 7 || tra_fin == 0)
    printf("Tendras que cursar programacion nuevamente\n");
  else
    printf("Aprobaste con la siguiente calificacion: %.2f\n",final);
  return(0);
}
```

Ejercicio 3

En un almacén se hace 20% de descuento a los clientes cuya compra supere los $1,000.00 ¿Cuál será la cantidad que pagará una persona por su compra?

```
# include <stdio.h>
main()
{
  float compra, desc, totcom;

  printf("Por Introduzca el valor de su compra: ");
  scanf("%f",&compra);

  if(compra >= 1000)
    desc = (compra*0.20);
  else
    desc = 0;

  totcom = (compra-desc);
  printf("El total de su compra es:%.2f\n",totcom);
  return(0);
}
```

Ejercicio 4

Realiza una calculadora que sea capaz de sumar, restar, multiplicar y dividir, el usuario debe escoger la operación e introducir los operandos.

```
#include <stdio.h>

void main(void)
{
  int opcion;
  float op1,op2;
  printf("1.SUMAR\n");
  printf("2.RESTAR\n");
  printf("3.DIVIDIR\n");
  printf("4.MULTIPLICAR\n");
  printf("5.SALIR\n");
  printf("Elegir opción: ");
  scanf("%d",&opcion);

  switch(opcion)
  {
    case 1:
 printf("Introduzca los operandos: ");
 scanf("%f %f",&op1,&op2);
  printf("Resultado: %.2f\n",(op1+op2));
    break;
```

```
        case 2:
    printf("Introduzca los operandos: ");
    scanf("%f %f",&op1,&op2);
     printf("Resultado: %.2f\n",(op1-op2));
        break;
        case 3:
    printf("Introduzca los operandos: ");
    scanf("%f %f",&op1,&op2);
     printf("Resultado: %.2f\n",(op1/op2));
        break;
        case 4:
    printf("Introduzca los operandos: ");
    scanf("%f %f",&op1,&op2);
     printf("Resultado: %.2f\n",(op1*op2));
        break;
        case 5:
    printf("Opción Salir\n");
        break;
    default:
        printf("Opción incorrecta");
     }
}
```

3.3. Estructura repetitiva

Estructuras repetitivas o iterativas

Las estructuras repetitivas o iterativas son aquellas en las que las acciones se ejecutan un número determinado o indeterminado de veces y dependen de un valor predefinido o el cumplimiento de una condición.

While

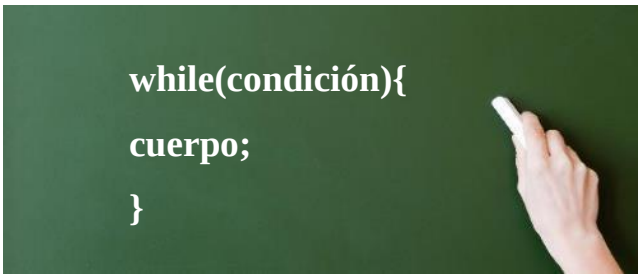
Ejecuta repetidamente el mismo bloque de código hasta que se cumpla una condición de terminación. Hay dos partes en un ciclo: *cuerpo* y *condición*.

Cuerpo

Es la sentencia o sentencias que se ejecutarán dentro del ciclo.

Condición

Es la condición de terminación del ciclo.



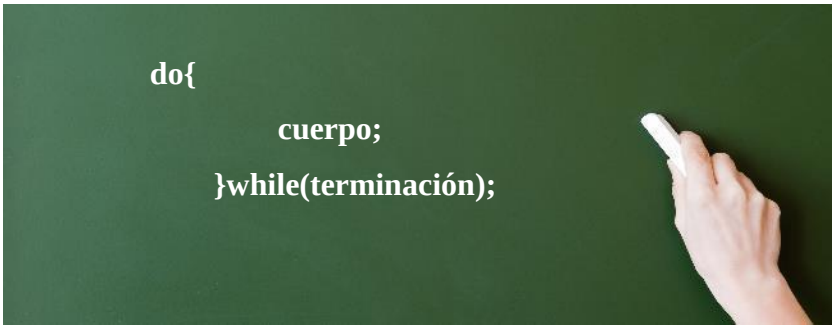
Ejemplo.

Este programa convierte una cantidad en yardas, mientras el valor introducido sea mayor que 0.

```
# include <stdio.h>
main()
{
  int yarda, pie, pulgada;
  printf("Por favor deme la longitud a convertir en yardas: ");
  scanf("%d",&yarda);
  while (yarda > 0)
  {
    pulgada = 36*yarda;
    pie  = 3*yarda;
    printf("%d yarda (s) = \n", yarda);
    printf("%d pie (s) \n", pie);
    printf("%d pulgada (s) \n", pulgada);
    printf("Por favor introduzca otra cantidad a \n");
    printf("convertir (0) para terminar el programa): ");
    scanf("%d",&yarda);
  }
  printf(">>> Fin del programa <<<");
  return(0);
}
```

do-while

Es lo mismo que en el caso anterior pero aquí como mínimo siempre se ejecutará el cuerpo al menos una vez, en el caso anterior es posible que no se ejecute ni una sola vez**.**



Ejemplo

Este es un ejemplo de un programa que utiliza un do-while para seleccionar una opción de un menú.

```
#include <stdio.h>
void menu(void);
main()
{
  menu();
  return(0);
}
void menu(void)
{
 char ch;
```

```
    printf("1. Opcion 1\n");
    printf("2. Opcion 2\n");
    printf("3. Opcion 3\n");
    printf("      Introduzca su opción: ");
    do {
      ch = getchar(); /* lee la selección desde el teclado */
      switch(ch) {
        case '1':
          printf("1. Opcion 1\n");
          break;
        case '2':
                        printf("2. Opcion 2\n");
          break;
        case '3':
                      printf("3. Opcion 3\n");
          break;
      }
    } while(ch!='1' && ch!='2' && ch!='3');
  }
```

Veamos más ejemplos:

Ejercicio 1

Realiza un programa que lea un variable e imprima su contenido, mientras el contenido de la variable sea distinto de cero.

```
# include <stdio.h>

main()
{
  int var;

  printf("Por introduzca un valor diferente de cero: ");
  scanf("%d",&var);
  while (var != 0)
  {
    printf("Contenido de la variable: %d\n", var);
    printf("Por favor introduzca otro valor\n");
    printf("(0) para terminar el programa): ");
    scanf("%d",&var);
  }
  printf(">>> Fin del programa <<<");
  return(0);
}
```

Ejercicio 2

Lee números negativos y conviértelos a positivos e imprime dichos números, mientras el número sea negativo.

```
# include <stdio.h>

main()
{
  int neg,res;

  printf("Por introduzca un valor negativo: ");
  scanf("%d",&neg);
  while (neg < 0)
  {
    res=neg*-1;
    printf("Valor positivo: %d\n",res);
    printf("Por favor introduzca otro valor\n");
    printf("Mayor o igual a (0) para terminar el programa): ");
    scanf("%d",&neg);
  }
  printf(">>> Fin del programa <<<");
  return(0);
}
```

Ejercicio 3

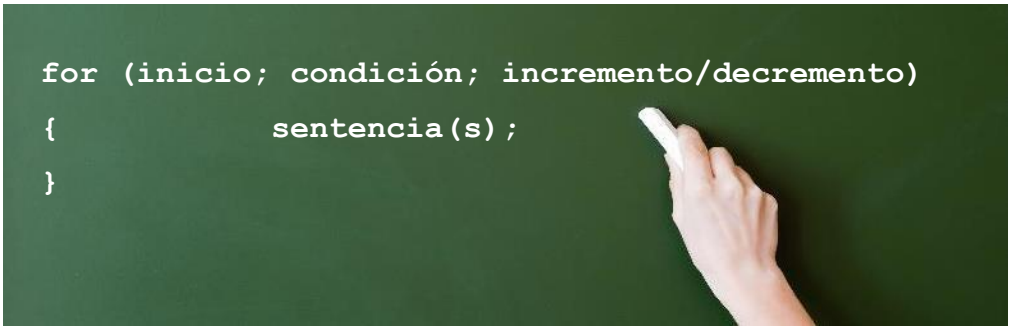
Realiza un programa que funcione como calculadora, que utilice un menú y un while.

```
#include <stdio.h>

#define SALIDA 0
#define BLANCO ' '

void main(void)
{
  float op1,op2;
  char operador = BLANCO;

  while (operador != SALIDA)
  {
    printf("Introduzca una expresión (a (operador) b): ");
    scanf("%f%c%f", &op1, &operador, &op2);

    switch(operador)
          {
      case '+':
                        printf("Resultado: %8.2f\n",(op1+op2));
%8.2f\n",(op1+op2));
        break;
      case '-':
                        printf("Resultado: %8.2f\n",(op1-op2));
        break;
      case '*':
                        printf("Resultado: %8.2f\n",(op1*op2));
        break;
```

```
                    case '/':
                                printf("Resultado: %8.2f\n",(op1/op2));
                      break;
                    case 'x':
                                operador = SALIDA;
                      break;
                        default: printf("Opción incorrecta");
                       }
                }
}
```

for

Realiza las mismas operaciones que en los casos anteriores pero la sintaxis es una forma compacta. Normalmente la condición para terminar es de tipo numérico. La iteración puede ser cualquier expresión matemática válida. Si de los 3 términos que necesita no se pone ninguno se convierte en un bucle infinito.



Ejemplo

Este programa muestra números del 1 al 100. Utilizando un bucle de tipo for.

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

```
void main(void)
{
  int n1=0;
  for (n1=1;n1<=20;n1++)
    printf("%d\n",n1);
  getch();
}
```

Veamos más ejemplos:

Ejercicio 1

Calcula el promedio de un alumno que tiene 7 calificaciones en la materia de Introducción a la programación.

```
#include<stdio.h>
void main(void)
{
int n1=0;
  float cal=0,tot=0;
  for (n1=1;n1<=7;n1++)
  {
        printf("Introduzaca la calificación %d:",n1);
        scanf(" %f",&cal);
      tot=tot+cal;
  }
  printf("La calificación es: %.2f\n",tot/7);
}
```

Ejercicio 2

Lee 10 números y obtén su cubo y su cuarta.

```
#include<stdio.h>
void main(void)
{
  int n1=0,num=0,cubo,cuarta;
  for (n1=1;n1<=10;n1++)
  {
    printf("Introduzca el numero %d:",n1);
    scanf(" %d",&num);
    cubo=num*num*num;
    cuarta=cubo*num;
    printf("El cubo de %d es: %d La cuarta es: %d\n",num,cubo,cuarta);
  }
}
```

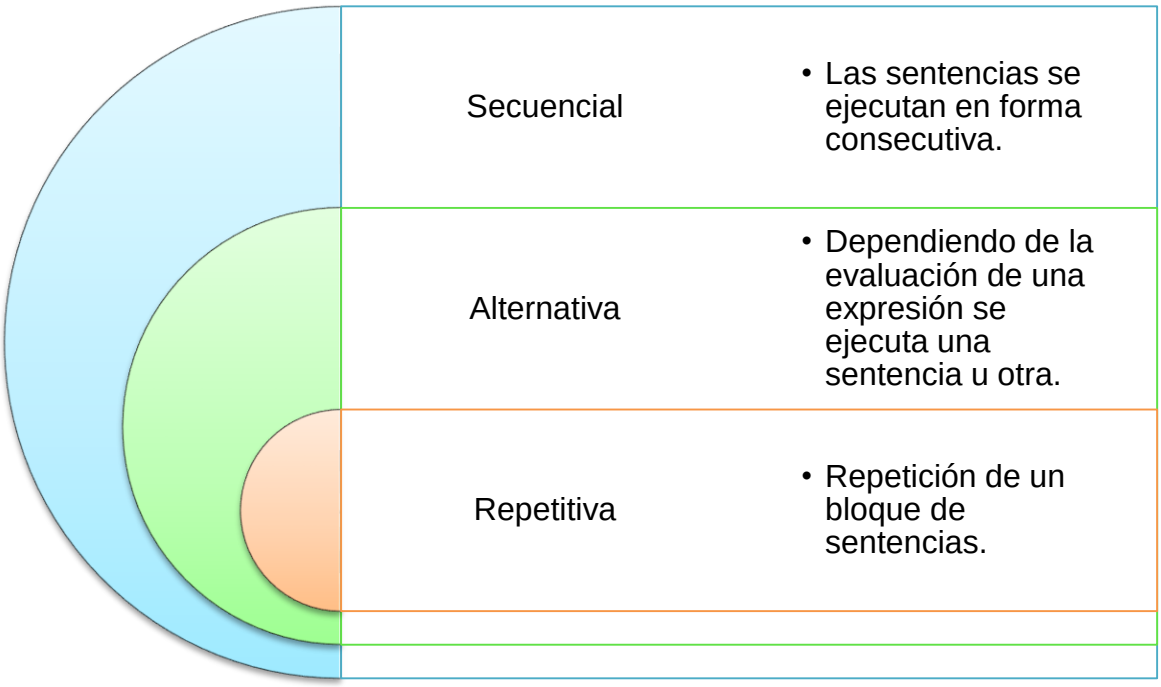
Ejercicio 3

Lee 10 números, e imprime solamente números positivos.

```
#include<stdio.h>
void main(void)
{
int n1=0,num=0;
for (n1=1;n1<=10;n1++)
{
printf("Introduzca el numero %d:",n1);
scanf(" %d",&num);
if (num > 0)
printf("El numero %d es positivo\n",num);
   }
}
```

RESUMEN

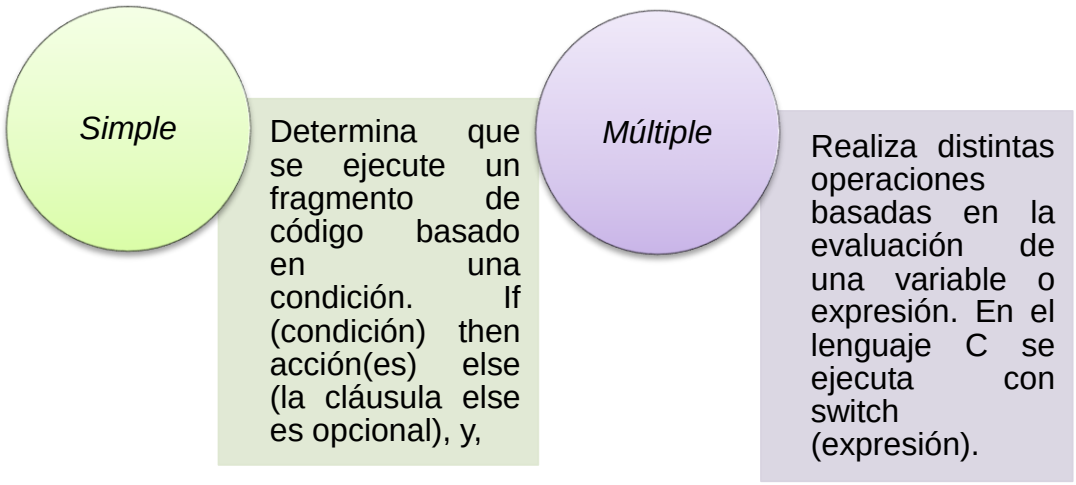
El teorema del programa estructurado indica que un programa puede escribirse empleando únicamente tres estructuras de control:



La programación estructurada tiene la ventaja de generar programas fáciles de entender y que los bloques de código sean auto-explicativos; su desventaja radica en que el mantenimiento se hace difícil con programas grandes y complejos.

Como se vio anteriormente, en la estructura secuencial, las instrucciones o acciones se ejecutan en forma sucesiva, es decir, una tras otra, sin omitir alguna, cada instrucción se ejecuta hasta que termina la anterior.

En cuanto a las estructuras alternativas, se puede decir hay dos tipos:



En la estructura repetitiva la ejecución de las acciones depende de un valor predefinido o del cumplimiento de una condición. Entre las estructuras repetitivas *while* y *do while;* la diferencia radica que, en la primera, la sentencia podría ejecutarse más de una sola vez, y en la segunda, se ejecuta cuando menos una vez. Otra estructura repetitiva es *for*, que realiza un número de iteraciones definido por sus términos: inicio, condición e incremento.

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Deitel (2004)	3	49-74
	4	89-116
Brian & Dennis (1991)	3	61-72

UNIDAD 4

Funciones

UNAM FCA
CEDIGEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Utilizar las funciones preconstruidas y podrá desarrollar sus propias funciones; identificará el alcance de las variables utilizadas y aplicará la recursividad.

TEMARIO DETALLADO

(18 horas)

4. Funciones

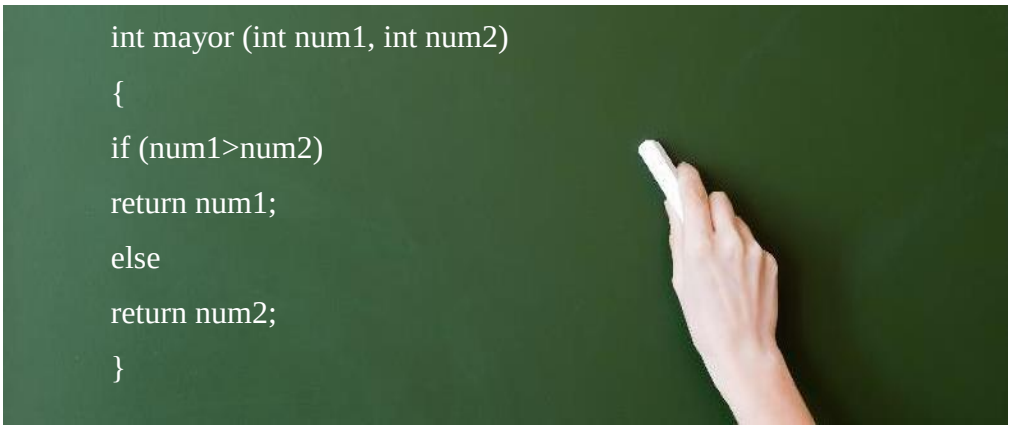
- 4.1. Internas
- 4.2. Definidas por el usuario
- 4.3. Ámbito de variables (locales y globales)
- 4.4. Recursividad

INTRODUCCIÓN

En el ámbito de la programación, **una función** es el término para describir una **secuencia de órdenes que hacen una tarea específica**.

- Un nombre único con el que se identifica y distingue a la función.
- Un valor que la función devolverá al terminar su ejecución.
- Una lista de parámetros que la función debe recibir para realizar su tarea.
- Un conjunto de órdenes o sentencias que debe ejecutar la función.

Observa en el siguiente ejemplo la estructura de una función con el nombre *mayor(),* que devolverá como resultado el número más grande de dos cantidades enteras positivas, que le son pasadas como parámetros:



Las funciones son las que realizan las tareas principales de un programa.

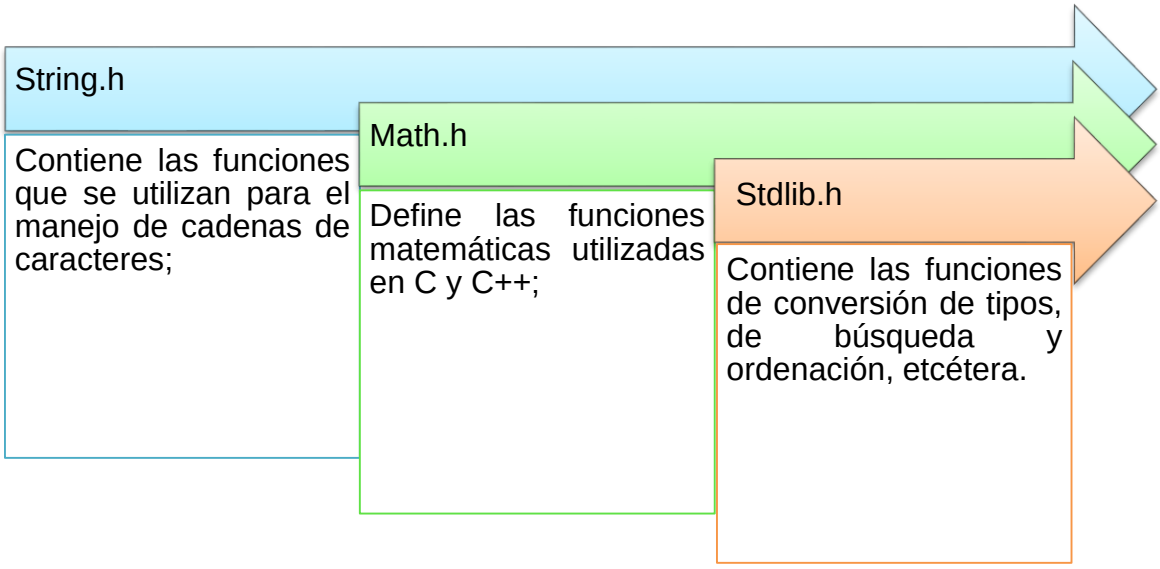
Las funciones son la característica más importante de C; son los bloques constructores; es el lugar donde se produce toda la actividad del programa. Las funciones realizan una tarea específica, como mandar a imprimir un mensaje en pantalla o abrir un archivo.

Subdivide en varias tareas el programa, así sólo se las tendrá que ver con diminutas piezas de programa, de pocas líneas, cuya escritura y corrección es una tarea simple. Las funciones pueden o no devolver y recibir valores del programa.

4.1. Internas

El lenguaje C cuenta con funciones internas que realizan tareas específicas. Por ejemplo, hay funciones para el manejo de caracteres y cadenas, matemáticas, de conversión, entre otras.

Dichas funciones están almacenadas dentro de bibliotecas de funciones[2] del lenguaje C, como por ejemplo:



Funciones de caracteres y cadenas

La biblioteca estándar de C tiene un variado conjunto de funciones de manejo de caracteres y cadenas. En una implementación estándar, las funciones de cadena requieren el archivo de cabecera string.h, que proporciona sus prototipos. Las funciones de caracteres utilizan ctype.h como archivo de cabecera.

[2] Para ahondar en las funciones de C y C++, remitirse al sitio: http://www.data-2013.cl/DOCS/INFORMATICA/PROGRC/cap-c9.html

FUNCIONES PARA EL MANEJO DE CARACTERES

isalpha	Devuelve un entero. DISTINTO DE CERO si la variable es una letra del alfabeto, en caso contrario devuelve cero. Cabecera: <ctype.h>.	**int isalpha(variable_char);**
isdigit	Devuelve un entero. DISTINTO DE CERO si la variable es un número (0 a 9), en caso contrario devuelve cero. Cabecera <ctype.h>.	**int isdigit(variable_char);**
isgraph	Devuelve un entero. DISTINTO DE CERO si la variable es cualquier carácter imprimible distinto de un espacio, si es un espacio CERO. Cabecera <ctype.h>.	**int isgraph(variable_char);**
islower	Devuelve un entero. DISTINTO DE CERO si la variable está en minúscula, en caso contrario devuelve cero. Cabecera <ctype.h>.	**int islower(variable_char);**
ispunct	Devuelve un entero. DISTINTO DE CERO si la variable es un carácter de puntuación, en caso contrario, devuelve cero. Cabecera <ctype.h>	**int ispunct(variable_char);**
isupper	Devuelve un entero. DISTINTO DE CERO si la variable está en mayúsculas, en caso contrario, devuelve cero. Cabecera <ctype.h>	**int isupper(varible_char);**

Veamos un ejercicio donde se utiliza la función isalpha.

Cuenta el número de letras y números que hay en una cadena. La longitud debe ser siempre de cinco, por no conocer aún la función que me devuelve la longitud de una cadena.

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
void main(void)
{
        int ind,cont_num=0,cont_text=0;
        char temp;
        char cadena[6];
        printf("Introducir 5 caracteres: ");
        gets(cadena);
        for(ind=0;ind<5;ind++)
        {
                temp=isalpha(cadena[ind]);
                if(temp)
                                        cont_text++;
                else
                        cont_num++;
        }
        printf("El total de letras es  %d\n",cont_text);
        printf("El total de números es %d",cont_num);
        getch();
}
```

Si la función isalpha devuelve un entero distinto de cero, se incrementa la variable cont_text, de lo contrario se incrementa la variable cont_num.

El ejercicio anterior muestra el uso de funciones para el manejo de caracteres, veamos las funciones para el manejo de cadenas.

FUNCIONES PARA EL MANEJO DE CADENAS

memset

- Inicializar una región de memoria (buffer) con un valor determinado. Se utiliza principalmente para inicializar cadenas con un valor determinado. Cabecera <string.h>.
  - **memset (var_cadena,'carácter',tamaño);**

strcat

- Concatena cadenas, es decir, añade la segunda a la primera, que no pierde su valor original. Hay que tener en cuenta que la longitud de la primera cadena debe ser suficiente para guardar la suma de las dos cadenas. Cabecera <string.h>.
  - **strcat(cadena1,cadena2);**

strchr

- Devuelve un puntero a la primera ocurrencia del carácter especificado en la cadena donde se busca. Si no lo encuentra, devuelve un puntero nulo. Cabecera <string.h>.
  - **strchr(cadena,'carácter');**
  - **strchr("texto",'carácter');**

strcmp

- Compara alfabéticamente dos cadenas y devuelve un entero basado en el resultado de la comparación. La comparación no se basa en la longitud de las cadenas. Muy utilizado para comprobar contraseñas. Cabecera <string.h>.
  - **strcmp(cadena1,cadena2);**
  - **strcmp(cadena2,"texto");**

RESULTADO	
VALOR	DESCRIPCIÓN
Menor a Cero	Cadena1 menor a Cadena2.
Cero	Las cadenas son iguales.

Mayor a Cero	Cadena1 mayor a Cadena2.

strcpy

- Copia el contenido de la segunda cadena en la primera. El contenido de la primera se pierde. Lo único que debemos contemplar es que el tamaño de la segunda cadena sea menor o igual al tamaño de la primera cadena. Cabecera <string.h>.
  - **strcpy(cadena1,cadena2);**
  - **strcpy(cadena1,"texto");**

strlen

- Devuelve la longitud de la cadena terminada en nulo. El carácter nulo no se contabiliza. Devuelve un valor entero que indica la longitud de la cadena. Cabecera <string.h>
  - **variable=strlen(cadena);**

strncat

- Concatena el número de caracteres de la segunda cadena en la primera. Ésta última no pierde la información. Hay que controlar que la longitud de la primera cadena tenga longitud suficiente para guardar las dos cadenas. Cabecera <string.h>.
  - **strncat(cadena1,cadena2,nº de caracteres);**

strncmp

- Compara alfabéticamente un número de caracteres entre dos cadenas. Devuelve un entero según el resultado de la comparación. Los valores devueltos son los mismos que en la función strcmp. Cabecera <string.h>.
  - **strncmp(cadena1,cadena2,nº de caracteres);**
  - **strncmp(cadena1,"texto",nº de caracteres);**

strncpy

- Copia un número de caracteres de la segunda cadena a la primera. En la primera cadena se pierden aquellos caracteres que se copian de la segunda. Cabecera <string.h>.
  - **strncpy(cadena1,cadena2,nº de caracteres);**
  - **strncpy(cadena1,"texto",nº de caracteres);**

strrchr

- Devuelve un puntero a la última ocurrencia del carácter buscado en la cadena. Si no lo encuentra devuelve un puntero nulo. Cabecera <string.h>.
  - **strrchr(cadena,'carácter');**
  - **strrchr("texto",'carácter');**

strpbrk

- Devuelve un puntero al primer carácter de la cadena que coincida con otro de la cadena a buscar. Si no hay correspondencia devuelve un puntero nulo. Cabecera <string.h>.
  - **strpbrk("texto","cadena_de_busqueda");**
  - **strpbrk(cadena,cadena_de_busqueda)**;

tolower

- Devuelve el carácter equivalente al de la variable en minúsculas si la variable es una letra; y no la cambia si la letra es minúscula. Cabecera <ctype.h>.
  - **variable_char=tolower(variable_char);**

toupper

- Devuelve el carácter equivalente al de la variable en mayúsculas si la variable es una letra; y no la cambia si la letra es minúscula. Cabecera <ctype.h>.
  - **variable_char=toupper(variable_char);**

Para ver ejemplos de ejercicios de cadenas, revisa los **Ejercicios de Cadenas** que a continuación se te muestran.

Ejercicio 1

Este programa busca la primera coincidencia y muestra la cadena a partir de esa coincidencia.

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

void main(void)
{
	char letra;
	char *resp;
	char cad[30];

	printf("Cadena: ");
	gets(cad);
	printf("Buscar Letra: ");
	letra=getchar();

	resp=strchr(cad,letra);

	if(resp)
		printf("%s",resp);
	else
		printf("No Esta");
	getch();
}
```

Ejercicio 2

Este programa busca la última coincidencia y muestra la cadena a partir de ese punto.

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

void main(void)
{
	char letra;
	char *resp;
	char cad[30];

	printf("Cadena: ");
	gets(cad);
	printf("Buscar Letra: ");
	letra=getchar();

	resp=strrchr(cad,letra);

	if(resp)
		printf("%s",resp);
	else
		printf("No Esta");

	getch();
}
```

Ejercicio 3

En este ejemplo se busca en una cadena a partir de un grupo de caracteres que introduce el usuario. Si encuentra alguna coincidencia, la imprime en pantalla, de lo contrario muestra un mensaje de error en pantalla.

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

void main(void)
{
        char letras[5];
        char *resp;
        char cad[30];
        printf("Introducir cadena: ");
gets(cad);
        printf("Posibles letras(4): ");
gets(letras);
        resp=strpbrk(cad,letras);
        if(resp)
                printf("%s",resp);
        else
                printf("Error");
        getch();
```

Funciones matemáticas

El estándar C define 22 funciones matemáticas que entran en las siguientes categorías, **trigonométricas**, **hiperbólicas**, **logarítmicas**, **exponenciales** y otras. Todas las funciones requieren el archivo de cabecera **math.h**.

acos	asin	atan
•Devuelve un tipo *double*. Muestra el arcocoseno de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>. •**double acos(variable_double);**	•Devuelve un tipo *double*. Muestra el arcoseno de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>. •**double asin(variable_double);**	•Devuelve un tipo *double*. Muestra el arcotangente de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>. •**double atan(variable_double);**

cos	sin	tan
• Devuelve un tipo *double*. Muestra el coseno de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera <math.h>. • **double cos(variable_double _radianes);**	• Devuelve un tipo *double*. Muestra el seno de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera <math.h>. • **Doublé sin(variable_double _radianes);**	• Devuelve un tipo *double*. Muestra la tangente de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera <math.h>. • **double tan(variable_double _radianes);**

cosh

- Devuelve un tipo *double*. Muestra el coseno hiperbólico de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.
  - **double cosh(variable_double);**

sinh

- Devuelve un tipo *double*. Muestra el seno hiperbólico de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.
  - **double sinh(variable_double);**

tanh

- Devuelve un tipo *double*. Muestra la tangente hiperbólica de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.
  - **double tanh(variable_double);**

ceil

- Devuelve un *double* que representa el menor entero sin serlo más que la variable redondeada. Por ejemplo, dado 1.02 devuelve 2.0. Si asignamos –1.02, devuelve –1. En resumen, redondea la variable a la alta. Cabecera <math.h>.
  - **double ceil(variable_double);**

floor

- Devuelve un *double* que representa el entero mayor, sin serlo más que la variable redondeada. Por ejemplo dado 1.02 devuelve 1.0. Si asignamos –1.2 devuelve –2. En resumen redondea la variable a la baja. Cabecera <math.h>.
  - **double floor(variable_double);**

fabs

- Devuelve un valor *float* o *double*. Devuelve el valor absoluto de una variable *float*. Se considera una función matemática, pero su cabecera es <stdlib.h>.
  - **var_float fabs(variable_float);**

labs

- Devuelve un valor *long*. Devuelve el valor absoluto de una variable long. Se considera una función matemática, pero su cabecera es <stdlib.h>.
  - **var_long labs(variable_long);**

abs

- Devuelve un valor entero. Devuelve el valor absoluto de una variable int. Se considera una función matemática, pero su cabecera es <stdlib.h>.
  - **var_float abs(variable_float);**

modf

- Devuelve un *double*. Descompone la variable en su parte entera y fraccionaria. La parte decimal es el valor que devuelve la función, su parte entera la guarda en el segundo término de la función. Las variables tienen que ser obligatoriamente de tipo *double*. Cabecera <math.h>.
  - **var_double_decimal= modf(variable,var_parte_entera);**

pow

- Devuelve un *double*. Realiza la potencia de un número base elevada a un exponente que nosotros le indicamos. Se produce un error si la base es cero o el exponente es menor o igual que cero. La cabecera es <math.h>.
  - **var_double=pow(base_double,exponente_double);**

sqrt

- Devuelve un *double*. Realiza la raíz cuadrada de la variable, ésta no puede ser negativa, si lo es se produce un error de dominio. La cabecera <math.h>.
  - **var_double=sqrt(variable_double);**

log

- Devuelve un *double*. Realiza el logaritmo natural (neperiano) de la variable. Se produce un error de dominio si la variable es negativa y un error de rango si es cero. Cabecera <math.h>.
  - **double log(variable_double);**

log10	ramdomize()	random
• Devuelve un valor *double*. Realiza el logaritmo decimal de la variable de tipo double. Se produce un error de dominio si la variable es negativa y un error de rango si el valor es cero. La cabecera que utiliza es <math.h>. • **double log10(var_double);**	• Inicializa la semilla para generar números aleatorios. Utiliza las funciones de tiempo para crear esa semilla. Esta función está relacionada con random. Cabecera <stdlib.h>. • **void randomize();**	• Devuelve un entero. Genera un número entre 0 y la variable menos uno. Utiliza el reloj de la computadora para ir generando esos valores. El programa debe llevar la función randomize para cambiar la semilla en cada ejecución. Cabecera <stdlib.h>. • **int random(variable_int);**

Ejemplo

Este programa imprime las diez primeras potencias de 10.

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

int main(void)
{
  double x = 10.0, y = 0.0;

  do {
    printf("%f\n", pow(x, y));
    y++;
  } while(y<11.0);

  return 0;
}
```

Como puede observarse en el programa, se utiliza la función **pow** para obtener la potencia de un número.

Funciones de conversión

En el estándar de C se definen funciones para realizar conversiones entre valores numéricos y cadenas de caracteres. La cabecera de todas estas funciones es **stdlib.h**. Se pueden dividir en dos grupos, conversión de valores numéricos a cadena y conversión de cadena a valores numéricos.

atoi	Devuelve un entero. Esta función convierte la cadena en un valor entero. La cadena debe contener un número entero válido, si no es así el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es <stdlib.h>.	**int atoi(variable_char);**
atol	Devuelve un *long*. Esta función convierte la cadena en un valor *long*. La cadena debe contener un número long válido, si no es así, el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es <stdlib.h>.	**long atol(variable_char);**
atof	Devuelve un *double*. Esta función convierte la cadena en un valor *double*. La cadena debe contener un número *double* válido, si no es así, el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es <stdlib.h>.	**double atof(variable_char);**

sprintf	Devuelve una cadena. Esta función convierte cualquier tipo numérico a cadena. Para convertir de número a cadena hay que indicar el tipo de variable numérica y tener presente que la longitud de la cadena debe poder guardar la totalidad del número. Admite también los formatos de salida, es decir, que se pueden coger distintas partes del número. La cabecera es <stdlib.h>.	**sprintf(var_cadena,"identificador",var_numerica);**
itoa	Devuelve una cadena. La función convierte un entero en su cadena equivalente y sitúa el resultado en la cadena definida en segundo término de la función. Hay que asegurarse de que la cadena sea lo suficientemente grande para guardar el número. Cabecera <stdlib.h>.	**itoa(var_entero,var_cadena,base);**

BASE	DESCRIPCIÓN
2	Convierte el valor en binario.
8	Convierte el valor a Octal.
10	Convierte el valor a decimal.
16	Convierte el valor a hexadecimal.

ltoa	Devuelve una cadena. La función convierte un long en su cadena equivalente y sitúa el resultado en la cadena definida en segundo término de la función. Hay que asegurarse de que la cadena sea lo suficientemente grande para guardar el número. Cabecera <stdlib.h>.	**ltoa(var_long,var_cadena,base) ;**

Existen otras bibliotecas en C, además de las vistas, por ejemplo:

Entrada y salida de datos (stdio.h).

- Como su nombre lo indica, se refiere a la entrada y salida estándar (E/S).

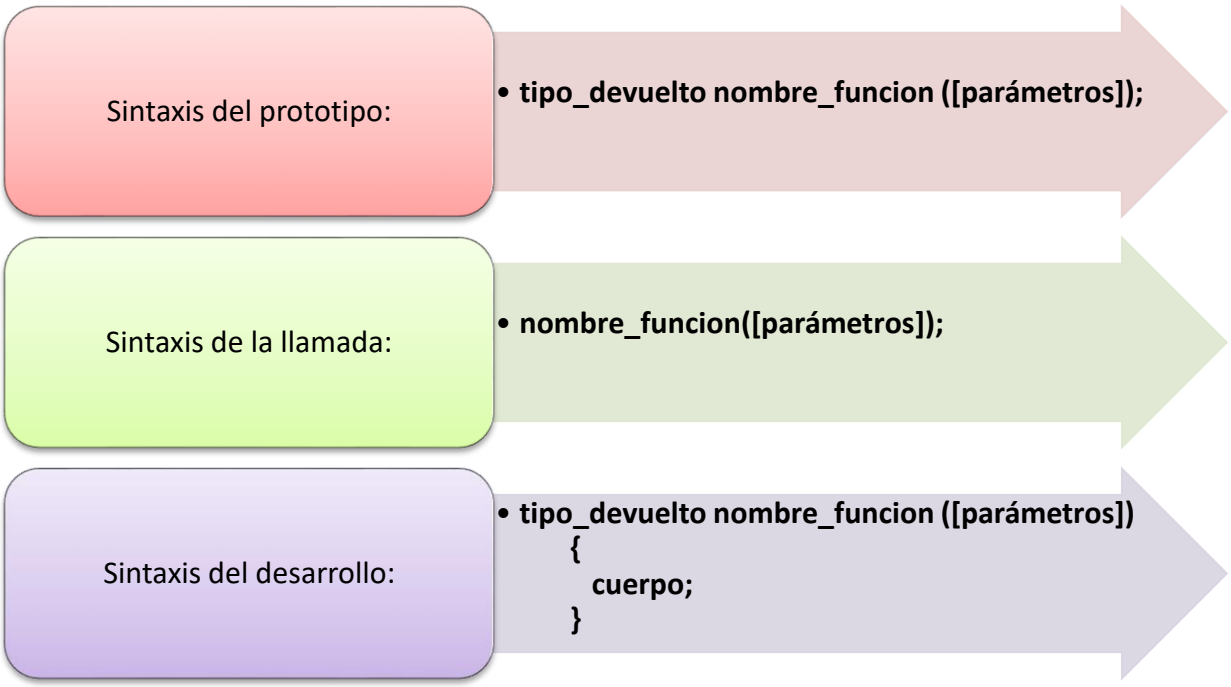
Memoria dinámica (alloc.h).

- Contiene funciones para asignar o liberar memoria u obtener información de bloques de memoria.

4.2. Definidas por el usuario

El mecanismo para trabajar con funciones es el siguiente: primero debemos declarar el prototipo de la función, a continuación debemos hacer la llamada y por último desarrollar la función. Los dos últimos pasos pueden cambiar, es decir, no es necesario que el desarrollo de la función esté debajo de la llamada.

Antes de seguir, debemos conocer las reglas de ámbito de las funciones. El código de una función es privado a ella, el código que comprende su cuerpo está oculto al resto del programa a menos que se haga a través de una llamada. Todas las variables que se definen dentro de una función son locales, con excepción de las variables estáticas.



Ejemplo

Un mensaje es desplegado a través de una función.

```
#include <stdio.h>
void saludo();
void primer_mensaje();
void main ( )
{
  saludo();
  primer_mensaje();
}
void saludo()
{
  printf ("Buenos días\n");
}
void primer_mensaje()
{
  printf("Un programa está formado ");
  printf("por funciones\n");
}
```

La función primer_mensaje despliega un mensaje en pantalla, esta función es llamada desde la función principal o main().

Cuando se declaran las funciones, es necesario informar al compilador el tamaño de los valores que se le enviarán y el tamaño del valor que retorna. En el caso de no indicar nada para el valor devuelto, toma por defecto el valor *int*.

Al llamar a una función se puede hacer la llamada por valor o por referencia. En el caso de hacerla por valor se copia el contenido del argumento al parámetro de la función, es decir, si se producen cambios en el parámetro, esto no afecta a los argumentos. C utiliza esta llamada por defecto. Cuando se utiliza la llamada por referencia lo que se pasa a la función es la dirección de memoria del argumento, por tanto, si se producen cambios, afectan también al argumento. La llamada a una función se puede hacer tantas veces como se quiera.

Ya sabes la forma en que se pueden pasar valores a las funciones, ahora observa los tipos de funciones que reciben o no parámetros y las que devuelven o no algún resultado:

Primer tipo

Las funciones de este tipo ni devuelven valor ni se les pasan parámetros. En este caso hay que indicarle que el valor que devuelve es de tipo void, y para indicarle que no recibirá parámetros también utilizamos el tipo void. Cuando realizamos la llamada no hace falta indicarle nada, se abren y cierran los paréntesis.

- **void nombre_funcion(void)**
- **nombre_funcion();**

Ejemplo

El siguiente programa es muy similar al anterior, este tipo de funciones no tienen parámetros ni devuelven ningún valor.

```
#include <stdio.h>
#include <conio.h>
void mostrar(void);
void main(void)
{
        printf("Estoy en la principal\n");
        mostrar();
        printf("De vuelta en la principal");
        getch();
}
void mostrar(void)
{
        printf("Ahora he pasado a la función, presione
cualquier tecla\n");
        getch();
}
```

Segundo tipo

Son funciones que devuelven un valor una vez que han terminado de realizar sus operaciones, sólo se puede devolver uno. La devolución se realiza mediante la sentencia return, que además de devolver un valor hace que la ejecución del programa vuelva al código que llamó a esa función. Al compilador hay que indicarle el tipo de valor que se va a devolver poniendo delante del nombre de la función el tipo por devolver. En este tipo de casos la función es como si fuera una variable, pues toda ella equivale al valor que devuelve.

- **tipo_devuelto nombre_funcion(void);**
- **variable=nombre_funcion();**

Ejemplo

Este programa calcula la suma de dos números introducidos por el usuario, la función suma() devuelve el resultado de la operación a la función principal a través de la función return().

```
#include <stdio.h>
#include <conio.h>
int suma(void);
void main(void)
{
        int total;
        printf("Suma valores\n");
        total=suma();
        printf("\n%d",total);
        getch();
}

int suma(void)
{
        int a,b,total_dev;
        printf("valores: ");
        scanf("%d %d",&a,&b);
        total_dev=a+b;
        return total_dev;
         }
```

Tercer tipo

En este tipo, las funciones pueden o no devolver valores, pero lo importante es que las funciones pueden recibir valores. Hay que indicar al compilador cuántos valores recibe y de qué tipo es cada uno de ellos. Se le indica poniéndolo en los paréntesis que tienen las funciones. Deben ser los mismos que en el prototipo.

- **void nombre_funcion(tipo1,tipo2…tipoN);**
- **nombre_funcion(var1,var2…varN);**

Ejemplo

Este programa utiliza una función de nombre resta(), que tiene dos parámetros, los parámetros son los operandos de una resta, además la función devuelve el resultado de la operación a la función principal a través de la función **return()**.

```
#include<stdio.h>

int resta(int x, int y); /*prototipo de la función*/

main()
{
        int a=5;
        int b=93;
        int c;

c=resta(a,b);
          printf("La diferencia es: %d\n",c);
return(0);
}

int resta(int x, int y) /*declaracion de la función*/
{
        int z;

        z=y-x;
return(z);      /*tipo devuelto por la función*/
        }
```

Para ver más ejercicios relacionados con este tema, revisa los siguientes:

Ejercicio 1

Realiza un programa que lea un número entero y determine si es par o impar.

```
/*Lee un numero entero y determina si es par o impar */
#include <stdio.h>
#define MOD %
/* %, es el operador que obtiene el resto de la división entera */
#define EQ ==
#define NE !=
#define SI 1
#define NO 0
void main ( )
{
        int n, es_impar(int);
        printf ("Introduzca un entero: \n");
        scanf ("%d", &n);
        if ( es_impar (n) EQ SI )
        printf ("El numero %d es impar. \n", n);
    else
        printf ("El numero %d no es impar. \n", n);
}

int es_impar (int x)
{
        int respuesta;
        if ( x MOD 2 NE 0 ) respuesta=SI;
        else
            respuesta=NO;
        return (respuesta);
 }
```

Ejercicio 2

Realiza un programa en C que sume, reste, multiplique y divida con datos introducidos por el usuario.

```
#include<stdio.h>

int resta(int x, int y); /*prototipo de la función*/
int suma(int x, int y);  /*prototipo de la función*/

main()
{
        int a,b,c,d;
        printf("Introduce el valor de a: ");
scanf("%d",&a);
        printf("\nIntroduce el valor de b: ");
scanf("%d",&b);
        c=resta(a,b);
d=suma(a,b);
        printf("La diferencia es: %d\n",c);
        printf("La suma es: %d\n",d);
return(0);
}

int resta(int x, int y) /*declaración de la función*/
{
        int z;
        z=y-x;
return(z);      /*tipo devuelto por la función*/
}

int suma(int x, int y) /*declaración de la función*/
{
        int z;
        z=y+x;
return(z);      /*tipo devuelto por la función*/
}
```

Ejercicio 3

Realiza una función que eleve un número al cubo.

```
#include<stdio.h>

int cubo(int base);
void main(void)
{
int res,num;
printf("Introduzca un numero:");
scanf(" %d",&num);
res=cubo(num);
printf("El cubo de %d es: %d\n",num,res);
}

int cubo(int x)
{
int cubo_res;
cubo_res=x*x*x;
return(cubo_res);
                }
```

Ejercicio 4

Realiza una función que eleve un número a un exponente cualquiera.

```
#include<stdio.h>

int cubo(int base,int expo);
void main(void)
{
int res,num,ex;
printf("Introduzca una base:");
scanf(" %d",&num);
printf("Introduzca un exponente:");
scanf(" %d",&ex);
res=cubo(num,ex);
printf("El numero %d elevado es: %d\n",num,res);
}

int cubo(int x, int y)
{
int i,acum=1;
for(i=1;i<=y;i++)
acum=acum*x;
return(acum);
                }
```

Ejercicio 5

Realiza una función que obtenga la raíz cuadrada de un número.

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

void impresion(void);

main()
{
	printf("Este programa extraerá una raiz cuadrada.\n\n");
	impresion();
	return(0);
}

void impresion(void)
{
	double z=4;
	double x;
	x=sqrt(z);
	printf("La raíz cuadrada de %lf es %lf \n",z,x);
}
```

Ejercicio 6

El método de la burbuja es un algoritmo de ordenación de los más sencillos, busca el arreglo desde el segundo hasta el último elemento comparando cada uno con el que le precede. Si el elemento que le precede es mayor que el elemento actual, los intercambia de tal modo que el más grande quede más cerca del final del arreglo. En la primera pasada, esto resulta en que el elemento más grande termina al final del arreglo.

El siguiente ejercicio usa un arreglo de diez elementos desordenados y utiliza la función bubble para ordenar los elementos. La función bubble compara un elemento con el siguiente, si es mayor, entonces los intercambia, para eso usa una variable temporal de nombre t. El proceso se repite un número de veces igual al número de elementos menos uno. El resultado final es un arreglo ordenado.

```
/* Programa de ordenamiento por burbuja */
#include <stdio.h>
int arr[10] = {3,6,1,2,3,8,4,1,7,2};
void bubble(int a[ ], int N);
int main(void)
{
      int i;
      putchar('\n');
      for (i = 0; i < 10; i++)
      {
            printf("%d ", arr[i]);
      }
      bubble(arr,10);
      putchar('\n');
      for (i = 0; i < 10; i++)
      {
      printf("%d ", arr[i]);
      }
      return 0;
}
```

```
void bubble(int a[], int N)
{
	int i, j, t;
	for (i = N-1; i >= 0; i--)
	{
		for (j = 1; j <= i; j++)
		{
			if (a[j-1] > a[j])
			{
				t = a[j-1];
				a[j-1] = a[j];
				a[j] = t;
			}
		}
	}
}
```

4.3. Ámbito de variables (locales y globales)

Variables locales

Las variables son **locales** cuando se declaran dentro de una función. Las variables locales sólo pueden ser referenciadas (utilizadas) por sentencias que estén dentro de la función donde han sido declaradas; cuando se sale de la función, los valores de estas variables se pierden.

Variables globales

Las variables son **globales** cuando son conocidas a lo largo de todo el programa, y se pueden usar desde cualquier parte del código. Mantienen sus valores durante toda la ejecución. Deben ser declaradas fuera de todas las funciones, incluida main(); sin embargo, al declarar variables locales dentro de la función main(); sus valores son reconocidos en todo el programa como si fueran variables globales, debido a que main() es la función principal.

4.4. Recursividad

Recursividad

La recursividad es el proceso de definir algo en términos de sí mismo, es decir, que las funciones pueden llamarse a sí mismas; esto se consigue cuando en el cuerpo de la función hay una llamada a la propia función, entonces se dice que es recursiva. Una función recursiva no hace una nueva copia de la función, sólo son nuevos los argumentos.

La principal ventaja de las funciones recursivas es que se pueden usar para crear versiones de algoritmos más claros y sencillos. Cuando se escriben funciones recursivas, se debe tener una sentencia *if* para forzar a la función a volver sin que se ejecute la llamada recursiva.

Ejemplo
Realizar un programa que obtenga la factorial de un número utilizando una función recursiva. Como podrás observar dentro de la función factorial, se hace una nueva llamada a la función, esto es una llamada recursiva.

```
#include <stdio.h>
double factorial(double respuesta);
main()
{
        double numero=3.0;
        double fact;
        fact=factorial(numero);
        printf("El factorial vale: %15.0lf \n",fact);
        return(0);
}
```

```
double factorial(double respuesta)
{
        if (respuesta <= 1.0)
                return(1.0);
        else
                return(respuesta*factorial(respuesta-1.0));
}
```

En el ejemplo anterior, la función trabaja de la siguiente manera:

- En cada iteración el valor de respuesta se reduce en 1.
- El valor de factorial es multiplicado por el valor de respuesta en cada iteración.
- En la primera iteración el valor de respuesta es 3, después 2 y por último 1.
- En la primera iteración se multiplica por (3-1), y después (2-1).

Observa el siguiente cuadro de valores de la función:

respuesta	respuesta*factorial(respuesta-1.0)	return
3.0	3.0 * factorial(3.0-1.0)	6.0
2.0	2.0 * factorial(2.0-1.0)	2.0
1.0		1.0

Las funciones recursivas pueden ahorrar la escritura de código, sin embargo, se deben usar con precaución, pues pueden generar un excesivo consumo de memoria.

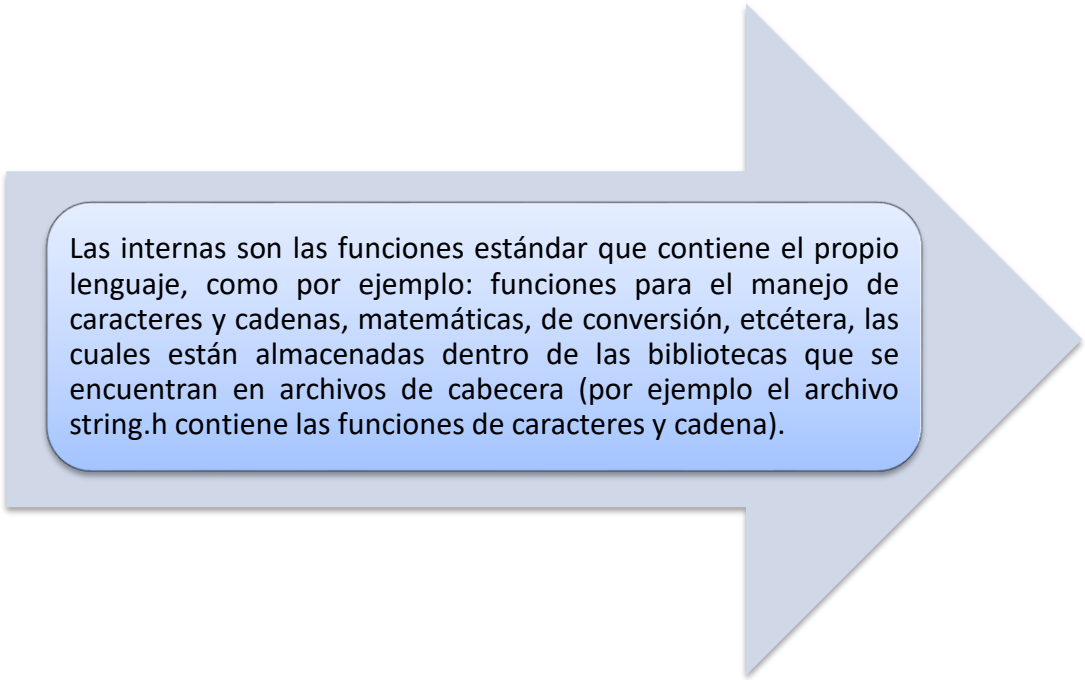
No se recomienda el uso de la recursividad en problemas de tipo d^N, en donde *d* representa la complejidad del programa y *N* el número de entradas al programa, es decir, que la complejidad se eleva exponencialmente a mayor número de entradas.

RESUMEN

En esta unidad se trataron las funciones internas y las definidas por el usuario, el ámbito de las variables y la recursividad.

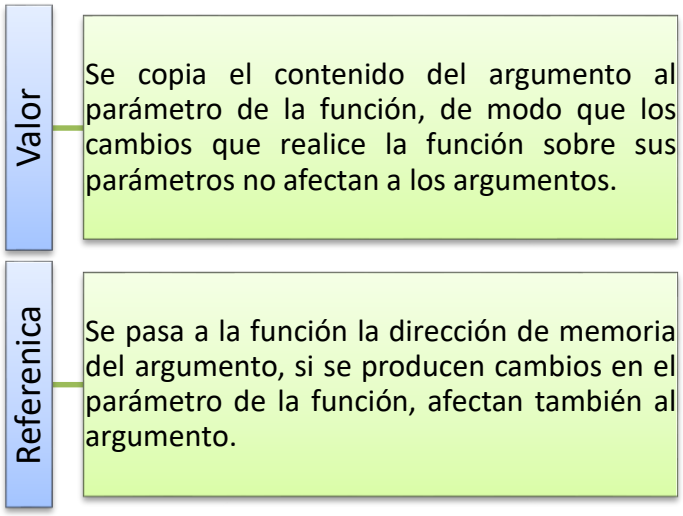
Se vio que una **función** es una secuencia de órdenes que hacen una tarea específica, subdividiendo al programa en tareas.

El lenguaje C cuenta con funciones **internas** y con funciones definidas por el usuario.



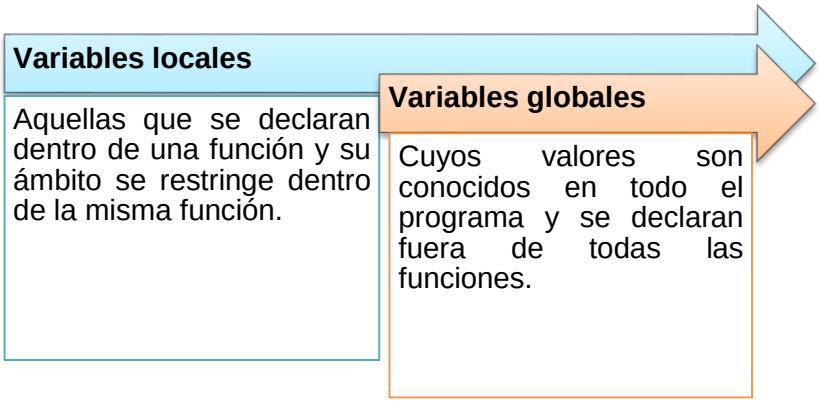
Para trabajar con las **funciones definidas por el usuario**, primero se tiene que declarar el prototipo de la función, luego su llamada y por último el desarrollo de la función.

Hay dos formas de llamadas que se hacen a una función:



Hay varios tipos de funciones, están aquellas que no devuelven valor ni se les pasan parámetros utilizando la palabra *void*; las que devuelven un valor mediante el uso de la sentencia *return;* y por último, las funciones que pueden o no devolver valores, pero que sí reciben parámetros.

Por otro lado, tenemos a las **variables locales y globales.**



En cuanto a la **recursividad**, nos referimos a ella como la capacidad que tiene una función de llamarse a sí misma, es decir, que dentro de la función hay una sentencia que realiza una llamada a la misma función pasándole nuevos argumentos.

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Joyanes (2006)	6	217-269

UNIDAD 5

Tipos de datos compuestos (estructuras)

UNAM FCA
CEDIGEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Podrá utilizar arreglos unidimensionales, multidimensionales y estructuras, para almacenar y procesar datos para aplicaciones específicas.

TEMARIO DETALLADO

(14 horas)

5. Tipos de datos compuestos (estructuras)

5.1. Arreglos Unidimensionales
5.2. Arreglos Multidimensionales
5.3. Arreglos y cadenas
5.4. Estructuras

INTRODUCCIÓN

Un arreglo es una colección de variables del mismo tipo que se referencian por un nombre en común. A un elemento específico de un arreglo se accede mediante un índice. Todos los arreglos constan de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento. Los arreglos pueden tener una o varias dimensiones.

Por otro lado, una estructura es una colección de variables que se referencia bajo un único nombre, y a diferencia del arreglo, puede combinar variables de tipos distintos.

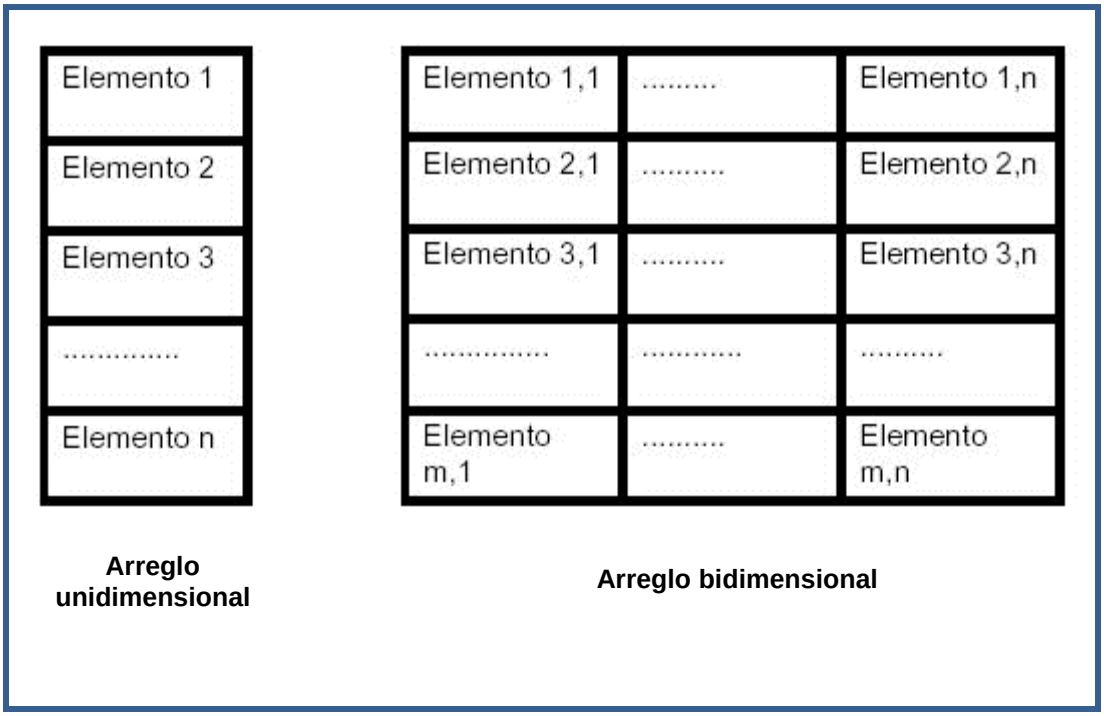
Tanto los arreglos como los registros son estructuras de datos que sirven para almacenar valores en memoria, la diferencia radica en que el arreglo solo te permite almacenar un tipo específico de datos: enteros, caracteres, fechas, etcétera, y los registros, como se ha indicado, admite diferentes tipos de datos, como clave de trabajador (entero), nombre (cadena), fecha de nacimiento (fecha), sueldo (flotante), etcétera.

5.1. Arreglos unidimensionales

Arreglos

Los **arreglos** son una colección de variables del mismo tipo que se referencian utilizando un nombre común. Un arreglo consta de posiciones de memoria contigua.

La dirección más baja corresponde al primer elemento, y la más alta al último. Un arreglo puede tener una o varias dimensiones. Para acceder a un elemento en particular de un arreglo se usa un índice.



El formato para declarar un arreglo unidimensional es:

Por ejemplo, para declarar un arreglo de enteros llamado *listanum* con diez elementos se hace de la siguiente forma:

```
int listanum[10];
```

Arreglo listanum[10]	
Posición	Valor almacenado
[0]	5
[1]	7
[2]	1
[3]	9
[4]	6
[5]	8
[6]	2
[7]	0
[8]	3
[9]	4

El arreglo listanum[10] tiene un tamaño de 10 elementos, es decir que puede almacenar hasta 10 números enteros en todas sus posiciones; en la tabla anterior, la primera

columna muestra, encerrado entre corchetes, el número de índice que identifica a la posición del arreglo, y en la segunda columna está el valor almacenado de la posición referida.

En C todos los arreglos usan cero como índice para el primer elemento. Por tanto, el ejemplo anterior declara un arreglo de enteros con diez elementos desde listanum[0] hasta listanum[9].

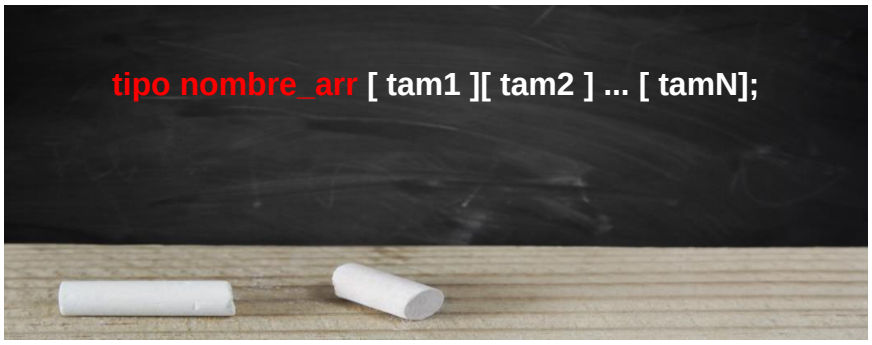
La forma en que se pueden acceder a los elementos de un arreglo es de la siguiente forma:

```
listanum[2] = 1; /* Asigna 1 al 3er elemento del arreglo listanum*/
```

```
num = listanum[2]; /* Asigna el contenido del 3er elemento a la variable num */
```

El lenguaje C realiza comprobación de contornos en los arreglos. En el caso de que sobrepase el final durante una operación de asignación, entonces no se asignarán valores a otra variable o a un trozo del código, esto es, si se dimensiona un arreglo de tamaño *N*, no se puede referenciar el arreglo por encima de *N* sin provocar un mensaje de error en tiempo de compilación o ejecución, provocando el fallo del programa. Como programador se es responsable de asegurar que todos los arreglos sean lo suficientemente grandes para guardar lo que pondrá en ellos el programa.

C permite arreglos con más de una dimensión, el formato general es:

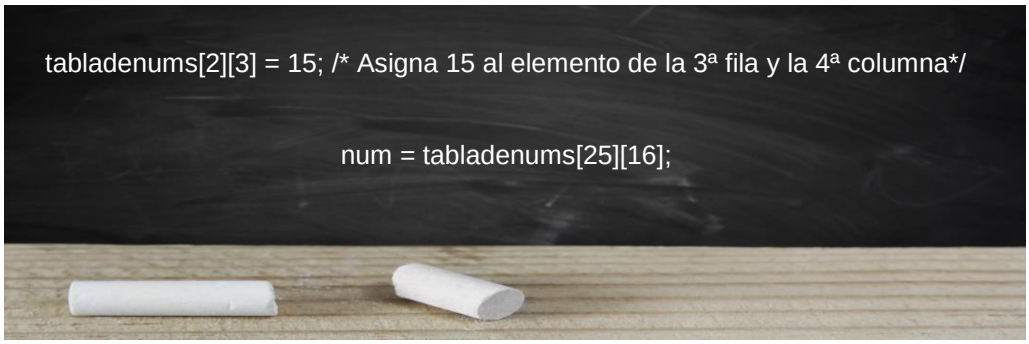


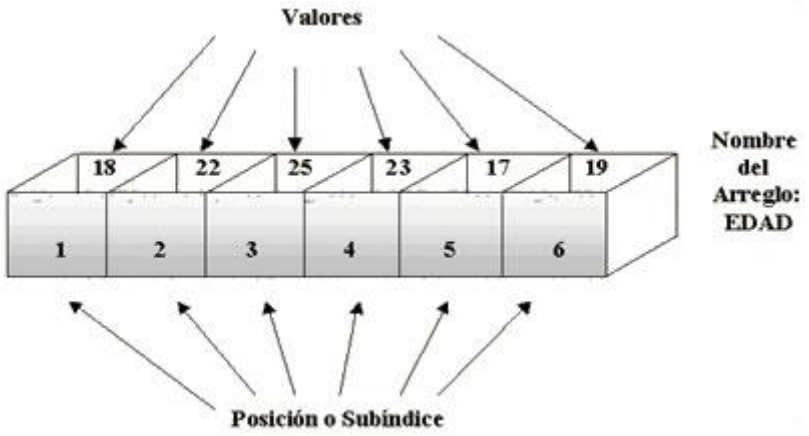
Por ejemplo, un arreglo de enteros bidimensionales se escribirá como:

```
int tabladenums[50][50];
```

Observar que, para declarar cada dimensión, lleva sus propios corchetes.

Para acceder a los elementos, se procede de forma similar al ejemplo del arreglo unidimensional, esto es,




Valores
18
22
25
23
17
19
1
2
3
4
5
6
Nombre del Arreglo: EDAD
Posición o Subíndice

SINTAXIS

tipo nombre_arreglo [nº elementos];

tipo nombre_arreglo [nº elementos]={valor1,valor2,valorN};

tipo nombre_arreglo[]={valor1,valor2,valorN};

INICIALIZACIÓN DE UN ELEMENTO

nombre_arreglo[indice]=valor;

UTILIZACIÓN DE ELEMENTOS

nombre_arreglo[indice];

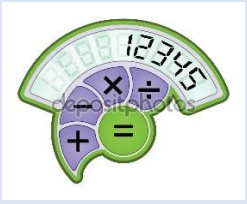
Ejercicio

Reserva 100 elementos enteros, inicializa todos y muestra el 5º elemento.

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

void main(void)
{
 int x[100];
 int cont;

 for(cont=0;cont<100;cont++)
        x[cont]=cont;

 printf("%d",x[4]);
 getch();
}
```

Revisa los ejercicios de arreglos unidimensionales para que puedas ver más ejemplos.

EJERCICIO 1



Veamos un ejercicio con arreglos utilizando números, el programa sumará, restará, multiplicará, dividirá los tres elementos de un arreglo denominado datos, y almacenará los resultados en un segundo arreglo denominado *resul*.

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

void main(void)
{
static int datos[3]={10,6,20}; //CARGA DEL ARREGLO datos

static int resul[5]={0,0,0,0,0}; //CARGA DEL ARREGLO resul
/* Las siguientes 5 líneas de código se carga el arreglo resul
con el resultado de las distintas operaciones aritméticas que
se efectúan con los valores del arreglo datos */


resul[0]=datos[0]+datos[2];
resul[1]=datos[0]-datos[1];
resul[2]=datos[0]*datos[2];
resul[3]=datos[2]/datos[0];
resul[4]=datos[0]%datos[1];

        /* el siguiente bloque de código se obtiene la salida
           en pantalla de los valores resultantes de las operaciones
           que están almacenados en el arreglo resul */
        printf("\nSuma: %d",resul[0]);
        printf("\nResta: %d",resul[1]);
        printf("\nMultiplicacion: %d",resul[2]);
        printf("\nDivision: %d",resul[3]);
        printf("\nResiduo: %d",resul[4]);
  }
```

La salida del programa quedaría como a continuación se indica:

```
Suma: 30
Resta: 4
Multiplicación: 200
División: 2
```

Residuo: 4

Quedando el arreglo *resul* con los siguientes valores:

Posición	Valor
resul[0]	30
resul[1]	4
resul[2]	200
resul[3]	2
resul[4]	4

EJERCICIO 2

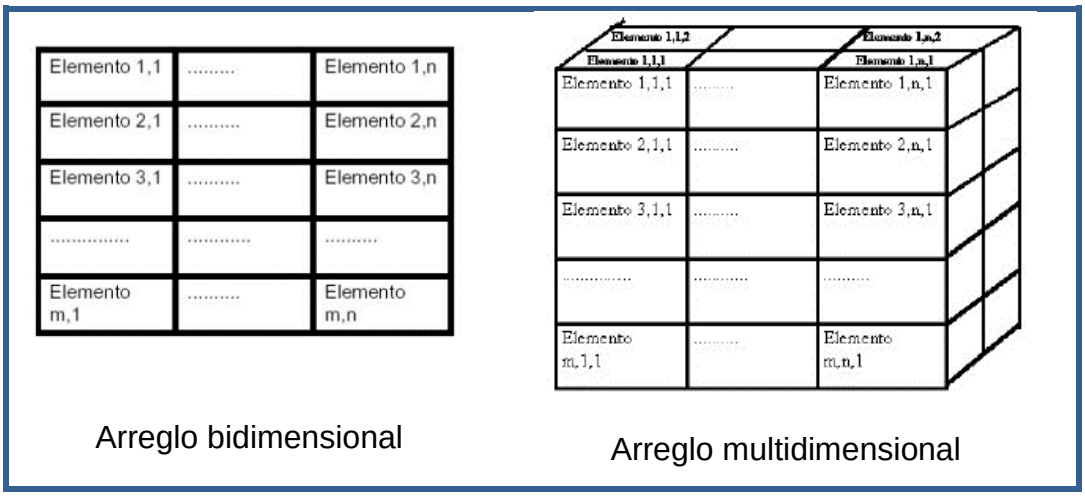
El siguiente ejercicio carga un arreglo de enteros con números desde el cero hasta el 99 y los imprime en pantalla.

```
#include <stdio.h>
int main(void)
{
        int x[100]; /* declara un arreglo de 100-integer */
        int t;
        /* carga x con valores de 0 hasta 99 */
        for(t=0; t<100; ++t)
x[t] = t;
        /* despliega el contenido de x */
        for(t=0; t<100; ++t)
printf("%d ", x[t]);
        return 0;
}
```

5.2. Arreglos multidimensionales

C admite arreglos multidimensionales, la forma más sencilla es la de los arreglos bidimensionales, éstos son esencialmente un arreglo de arreglos unidimensionales, se almacenan en matrices fila-columna, el primer índice muestra la fila y el segundo la columna. Esto significa que el índice más a la derecha cambia más rápido que el de más a la izquierda cuando accedemos a los elementos.

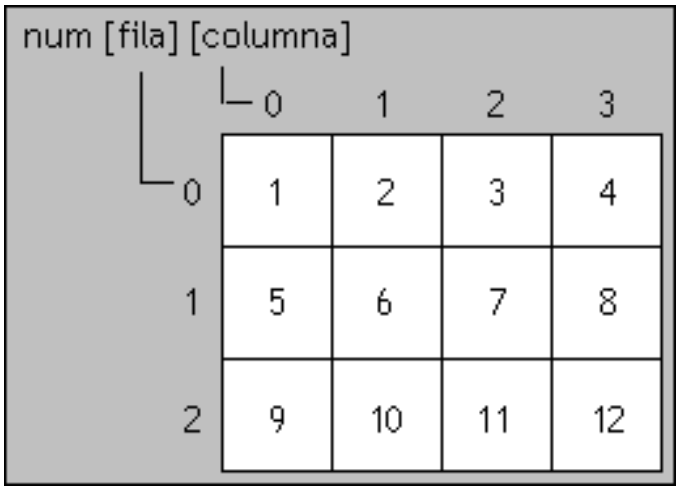
Los arreglos multidimensionales nos sirven para almacenar tablas de valores que tengan varias filas y columnas, e inclusive profundidad (cubos de datos), así podemos hacer referencia a un dato indicando su posición, fila y columna, o bien, fila, columna, profundidad, en su caso.



Arreglo bidimensional

Arreglo multidimensional

SINTAXIS	tipo nombre_arreglo[fila][columna];
	tipo nomb_arreglo[fil][col]={{v1,v2,vN},{v1,v2,vN},{vN}};
	tipo nomb_arreglo[][]={{v1,v2,vN},{v1,v2,vN},{vN}};



Inicialización de un elemento	Utilización de un elemento
•nombre_arreglo[indice_fila][indice_columna]=valor;	•nombre_arreglo[indice_fila][indice_columna];

Para conocer el tamaño que tiene un array bidimensional tenemos que multiplicar las filas por las columnas, por el número de bytes que ocupa en memoria el tipo del array. Es exactamente igual que con los array unidimensionales lo único que se añade son las columnas.

filas * columnas * bytes_del_tipo

Ejercicio 1

Ejemplo de un arreglo de dos dimensiones. El programa utiliza un arreglo de dos dimensiones y lo llena con el valor de 1.

```
# include <stdio.h>
#define R 5
#define C 5
main()
{
        int matriz[R][C];
int i,j;
        for(i=0;i<R;i++)
                for(j=0;j<C;j++)
                matriz[i][j] = 1;
for(i=0;i<R;i++)
                for(j=0;j<C;j++)
                        printf("%d\n",matriz[i][j]);
                for(i=0;i<R;i++)
                {
                        printf("NUMERO DE RENGLON: %d\n",i);
                        for(j=0;j<C;j++)
                                printf("%d ",matriz[i][j]);
                        printf("\n\n");
                }
printf("Fin del programa");
        return(0);
}
```

Ejercicio 2

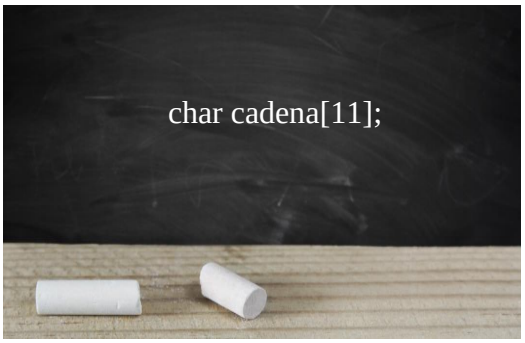
El siguiente ejercicio suma dos matrices.

```
#include <stdio.h>
void main(void)
{
        int matrix1[2][2];
        int matrix2[2][2];
        int res[2][2];

        printf("Introduzca los cuatro valores de la matriz 1: ");
        scanf("%d %d %d %d",
        &matrix1[0][0],&matrix1[0][1],&matrix1[1][0],&matrix1[1][1]);

         printf("Introduzca los cuatro valores de la matriz 2: ");
        scanf("%d %d %d %d",
&matrix2[0][0],&matrix2[0][1],&matrix2[1][0],&matrix2[1][1]);

         res[0][0] = matrix1[0][0] + matrix2[0][0];
       res[0][1] = matrix1[0][1] + matrix2[0][1];
       res[1][0] = matrix1[1][0] + matrix2[1][0];
       res[1][1] = matrix1[1][1] + matrix2[1][1];

         printf("%d %d\n",res[0][0],res[0][1]);

         printf("%d %d\n",res[1][0],res[1][1]);
```

5.3. Arreglos y cadenas

El uso más común de los arreglos es su utilización con **cadenas**, conjunto de caracteres terminados por el carácter nulo (‘\0’). Por tanto, cuando se quiera declarar un arreglo de caracteres se pondrá siempre una posición más. No es necesario añadir explícitamente el carácter nulo, el compilador lo hace automáticamente.

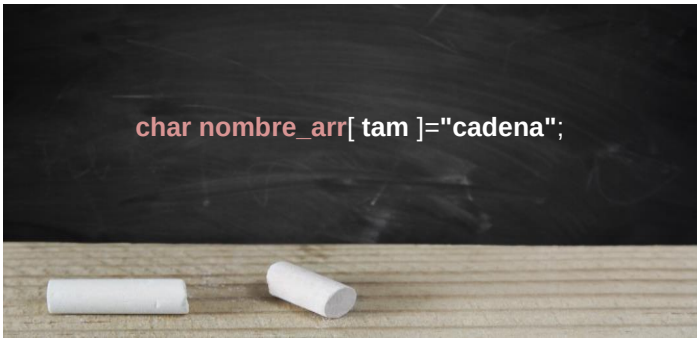
A diferencia de otros lenguajes de programación que emplean un tipo denominado cadena *string* para manipular un conjunto de símbolos, en C, se debe simular mediante un arreglo de caracteres, en donde la terminación de la cadena se debe indicar con nulo. Un nulo se especifica como '\0'. Por lo anterior, cuando se declare un arreglo de caracteres se debe considerar un carácter adicional a la cadena más larga que se vaya a guardar. Por ejemplo, si se quiere declarar un arreglo cadena que guarde una cadena de diez caracteres, se hará como:



Por ejemplo, para almacenar la frase “Hola mundo” que tiene 10 caracteres (contando también el espacio), el arreglo se debe declarar con 11 espacios para quedar como sigue:

0	1	2	3	4	5	6	7	8	9	10
H	o	l	a		m	u	n	d	o	\0

Se pueden hacer también inicializaciones de arreglos de caracteres en donde automáticamente C asigna el carácter nulo al final de la cadena, de la siguiente forma:



Por ejemplo, el siguiente fragmento inicializa cadena con ``hola'':

```
char cadena[5]="hola";
```

El código anterior es equivalente a:

```
char cadena[5]={'h','o','l','a','\0'};
```

SINTAXIS	char nombre[tamaño];
	char nombre[tamaño]=”cadena”;
	char nombre[]=”cadena”;
	char nombre[]=’c’;

Ejercicio

Realiza un programa en C en el que se representen los tres estados del agua con una cadena de caracteres. El arreglo agua_estado1 debe ser inicializado carácter por carácter con el operador de asignación, ejemplo: agua_estado1[0] = 'g'; el arreglo agua_estado2 será inicializado utilizando la función **scanf**; y el arreglo agua_estado3 deberá ser inicializado de una sola, utiliza la palabra **static**.

```
/* Ejemplo del uso de arrays y cadenas de caracteres */
#include <stdio.h>
main()
{
char agua_estado1[4],                  /* gas */
agua_estado2[7];                  /* solido */
static char agua_estado3[8] = "liquido";  /* liquido */
agua_estado1[0] = 'g';
agua_estado1[1] = 'a';
agua_estado1[2] = 's';
agua_estado1[3] = '\0';
printf("\n\tPor favor introduzca el estado del agua -> solido ");
scanf("%s",agua_estado2);
printf("%s\n",agua_estado1);
printf("%s\n",agua_estado2);
printf("%s\n",agua_estado3);
return(0);
}
```

5.4. Estructuras

C proporciona formas diferentes de creación de tipos de datos propios. Uno de ellos es la agrupación de variables bajo un mismo nombre, otra es permitir que la misma parte de memoria sea definida como dos o más tipos diferentes de variables y también crear una lista de constantes enteras con nombre.

Estructura		
Colección de variables que se referencia bajo un único nombre, proporcionando un medio eficaz de mantener junta una información relacionada.	Las variables que componen la estructura se llaman miembros de la estructura y están relacionadas lógicamente con las otras.	Otra característica es el ahorro de memoria y evitar declarar variables que técnicamente realizan las mismas funciones.

Una *definición de estructura* forma una plantilla que se puede usar para crear variables de estructura. Las variables que forman la estructura son llamados *elementos estructurados*.

Generalmente, todos los elementos en la estructura están relacionados lógicamente unos con otros. Por ejemplo, se puede representar una lista de nombres de correo en una estructura. Mediante la palabra clave *struct* se le indica al compilador que defina una plantilla de estructura.

```
struct direc
{
   char nombre[30];
   char calle[40];
   char ciudad[20];
   char estado[3];
   unsigned int codigo;
};
```

Con el trozo de código anterior *no ha sido declarada ninguna variable*, tan sólo se ha definido el formato. Para declarar una variable, se hará del modo siguiente:

```
struct direc info_direc;
```

En la variable de tipo estructura *info_direc* podrían almacenarse los datos del registro de una persona como a continuación se muestra:

Nombre: Juan Carlos Mendoza
Calle: Pedro Enríquez Ureña
Ciudad: Ciudad de México
Estado: Distrito Federal
Código: 09990

SINTAXIS

```
struct nombre{
var1;
var2;
varN;
};
.
struct nombre etiqueta1,etiquetaN;
```

Los miembros individuales de la estructura se referencian utilizando la etiqueta de la estructura, el operador punto(.) y la variable a la que se hace referencia. Los miembros de la estructura deben ser inicializados fuera de ella, si se hace en el interior da error de compilación.

```
etiqueta.variable;
```

Ejercicio

El siguiente programa almacena información acerca de automóviles, debido a que se maneja información de distintos tipos, sería imposible manejarla en un arreglo, por lo que es necesario utilizar una estructura.

El programa le pide al usuario que introduzca los datos necesarios para almacenar información de automóviles, cuando el usuario termina la captura, debe presionar las teclas ctrl.+z.

```
/* programa en C que muestra la forma de crear una estructura */


# include <stdio.h>

struct coche {
   char fabricante[15];
   char modelo[15];
   char matricula[20];
   int antiguedad;
   long int kilometraje;
   float precio_nuevo;
   } miauto;

main()
{
   printf("Introduzca el fabricante del coche.\n");
   gets(miauto.fabricante);
   printf("Introduzca el modelo.\n");
   gets(miauto.modelo);
   printf("Introduzca la matricula.\n");
```

```
  gets(miauto.matricula);
  printf("Introduzca la antiguedad.\n");
  scanf("%d",&miauto.antiguedad);
  printf("Introduzca el kilometraje.\n");
  scanf("%ld",&miauto.kilometraje);
  printf("Introduzca el precio.\n");
  scanf("%f",&miauto.precio_nuevo);
  getchar(); /*vacia la memoria intermedia del teclado*/

  printf("\n\n\n");
  printf("Un %s %s con %d años de antiguedad con número de matricula
#%s\n",miauto.fabricante,miauto.modelo,miauto.antiguedad,miauto.matricula);
  printf("actualmente con %ld kilómetros",miauto.kilometraje);
  printf(" y que fue comprado por $%5.2f\n",miauto.precio_nuevo);
  return(0);
}
```

Por favor revisa los ejercicios de estructuras para que puedas ver más ejemplos.

EJERCICIO 1

Realiza un programa que utilice la estructura anterior, pero que se encuentre dentro de un ciclo *for* para que el usuario pueda determinar cuántos autos va a introducir.

```
/* programa en C que muestra la forma de crear una estructura */

# include <stdio.h>

int i,j;
struct coche {
  char fabricante[15];
  char modelo[15];
  char matricula[20];
  int antiguedad;
  long int kilometraje;
  float precio_nuevo;
  } miauto;

main()
{
  printf("Introduce cuantos autos vas a capturar:\n");
```

```
  scanf("%d",&j);
  flushall(); /*vacia la memoria intermedia del teclado */
  for(i=1;i<=j;i++)
  {
    printf("Introduzca el fabricante del coche.\n");
    gets(miauto.fabricante);
    printf("Introduzca el modelo.\n");
    gets(miauto.modelo);
    printf("Introduzca la matricula.\n");
    gets(miauto.matricula);
    printf("Introduzca la antiguedad.\n");
    scanf("%d",&miauto.antiguedad);
    printf("Introduzca el kilometraje.\n");
    scanf("%ld",&miauto.kilometraje);
    printf("Introduzca el precio.\n");
    scanf("%f",&miauto.precio_nuevo);
    getchar(); /*vacia la memoria intermedia del teclado*/

    printf("\n\n\n");
    printf("Un %s %s con %d años de antiguedad con número de matricula
#%s\n",miauto.fabricante,miauto.modelo,miauto.antiguedad,miauto.matricula);
    printf("actualmente con %ld kilómetros",miauto.kilometraje);
    printf(" y que fue comprado por $%5.2f\n",miauto.precio_nuevo);
  flushall();
  }
  return(0);
}
```

EJERCICIO 2

Veamos ahora un ejemplo con arreglos, cadenas y estructuras.

```
/* Inicialización y manejo de "arreglos", cadenas y estructuras.*/
# include <stdio.h>
void main()
{
int i, j;
static int enteros [5] = { 3, 7, 1, 5, 2 };
static char cadena1 [16] = "cadena";
static char cadena2 [16] = { 'c','a','d','e','n','a','\0' };
static int a[2][5] = {
{ 1, 22, 333, 4444, 55555 },
{ 5, 4, 3, 2, 1 }
```

```
};
static int b[2][5] = { 1,22,333,4444,55555,5,4,3,2,1 };
static char *c = "cadena";
static struct {
int i;
float x;
} sta = { 1, 3.1415e4}, stb = { 2, 1.5e4 };
static struct {
char c;
int i;
float s;
} st [2][3] = {
{{ 'a', 1, 3e3 }, { 'b', 2, 4e2 }, { 'c', 3, 5e3 }},
{ { 'd', 4, 6e2 },}
};
printf ("enteros:\n");
for ( i=0; i<5; ++i ) printf ("%d ", enteros[i]);
printf ("\n\n");
printf ("cadena1:\n");
printf ("%s\n\n", cadena1);
printf ("cadena2:\n");
printf ("%s\n\n", cadena2);
printf ("a:\n");
for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d ", a[i][j]);
printf("\n\n");
printf ("b:\n");
for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d ", b[i][j]);
printf ("\n\n");
printf ("c:\n");
printf ("%s\n\n", c);
printf ("sta:\n");
printf ("%d %f \n\n", sta.i, sta.x);
printf ("st:\n");
for (i=0; i<2; ++i) for (j=0; j<3; ++j)
printf ("%c %d %f\n", st[i][j].c, st[i][j].i, st[i][j].s);
}
```

La salida del anterior programa es la siguiente:

enteros:

```
3 7 1 5 2
cadena1:
cadena
```

```
cadena2:
cadena
a:
1 22 333 4444 55555 5 4 3 2 1
b:
1 22 333 4444 55555 5 4 3 2 1
c:
cadena
sta:
1 31415.000000
st:
a 1 3000.000000
b 2 400.000000
c 3 5000.000000
d 4 600.000000
0 0.000000
0 0.000000
```

RESUMEN

En esta unidad se abordaron las características más importantes de los arreglos unidimensionales y multidimensionales, así como la relación de los arreglos y la gestión de cadenas. Por último, se trató el tema de las estructuras, una estructura es un conjunto de variables de distinto tipo.

Un arreglo es una colección de variables del mismo tipo que se referencian por un nombre común. Los registros son una colección de variables de distinto tipo, referenciadas bajo un mismo nombre. Ambas son estructuras de datos que sirven para almacenar valores referenciados en memoria.

En C hay arreglos unidimensionales que están formados por varias filas y una sola columna, y arreglos muldimensionales que son matrices fila-columna, de los cuales su forma más sencilla es la de los arreglos bidimensionales. Ambos sirven para almacenar valores en memoria.

La utilización de cadenas es el uso más común que se les puede dar a los arreglos, y en C se debe simular mediante un arreglo de caracteres, en donde la terminación de la cadena se indica con un nulo.

Una estructura es una plantilla que se puede usar para crear variables de estructura. Las variables que la componen se llaman miembros de la estructura y están relacionados lógicamente unos con otros.

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Joyanes (2005)	7	275-291
	8	309-325

UNIDAD 6

Manejo de apuntadores

UNAM FCA
CEDIGEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Utilizará apuntadores en aplicaciones con arreglos, estructuras y funciones y podrá hacer uso dinámico de la memoria.

TEMARIO DETALLADO

(8 horas)

6. Manejo de apuntadores

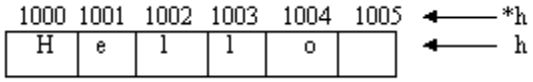
6.1. Introducción a los apuntadores
6.2. Apuntadores y arreglos
6.3. Apuntadores y estructuras
6.4. Apuntadores y funciones
6.5. Manejo dinámico de memoria

INTRODUCCIÓN

Apuntador

Es una variable que contiene una dirección de memoria. El uso de los apuntadores permite el manejo de la memoria de una manera más eficiente, los apuntadores son muy utilizados para el almacenamiento de memoria de manera dinámica.

Observa la siguiente figura:



La variable **h* representa las direcciones de memoria en donde están almacenados los caracteres contenidos en la variable de cadena *h*.

El lenguaje C usa los apuntadores en forma extensiva, en ocasiones es la única forma de expresar algunos cálculos. Con el uso de apuntadores se genera código compacto y eficiente convirtiendo a C en una herramienta poderosa.

C usa apuntadores explícitamente con arreglos, estructuras y funciones. Los arreglos almacenan valores, pero cuando se usa un arreglo de apuntadores, lo que se guarda en el arreglo son direcciones de memoria.

Los apuntadores también se utilizan para referenciar estructuras y poder pasar sus direcciones de memoria a las funciones.

El manejo dinámico de memoria se refiere a la capacidad que tiene un programa para obtener memoria en tiempo de ejecución asignando o liberando memoria asignada según lo vaya requiriendo el programa.12

6.1. Introducción a los apuntadores

Un apuntador o puntero es una variable que contiene una dirección de memoria. Esa dirección es la posición de un objeto (normalmente una variable) en memoria. Si una variable va a contener un puntero, entonces tiene que declararse como tal. Cuando se declara un puntero que no apunte a ninguna posición válida ha de ser asignado a un valor nulo (un cero). Se pueden tener apuntadores en cualquier tipo de variable.

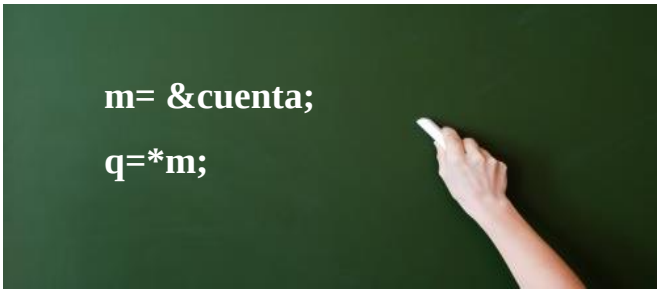
Para declarar un apuntador se tiene:

SINTAXIS	**tipo *nombre;**

Operador

El operador unario o monádico **&** devuelve la dirección de memoria de la variable de su operando. El operador de referencia ***** devuelve el valor de la variable referenciada por un apuntador.

Observa el siguiente ejemplo:



La dirección de memoria de la variable cuenta corresponde a m. La dirección no tiene nada que ver con el valor de cuenta. En la segunda línea, pone el valor de cuenta en q.

Ejercicio 1

El siguiente programa obtiene la dirección de memoria de una variable introducida por el usuario.

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

void main(void)
{
        int *p;
        int valor;

        printf("Introducir valor: ");
        scanf("%d",&valor);

        p=&valor;

        printf("Dirección de memoria de valor es: %p\n",p);
        printf("El valor de la variable es: %d",*p);
        getch();
}
```

El siguiente ejercicio (2) utiliza dos apuntadores, en el apuntador p_1 se guarda la dirección de memoria de la variable x, y en la variable p_2 se asigna el contenido del apuntador p_1, por último se imprime el contenido del apuntador p_2.

Ejercicio 2

```
#include <stdio.h>
void main(void)
{
        int x;
        int * p1, *p2;
        p1=&x;
        p2= p1;
        printf("%p",p2);
}
```

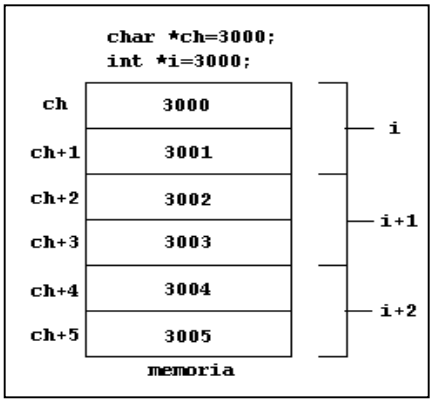
Aritmética

Las dos operaciones aritméticas que se pueden usar como punteros son la suma y la resta. Cuando se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base. Cada vez que se reduce, apunta a la posición del elemento anterior.

p_1++;

p_1--;

p_1+12;



Comparación

Se pueden comparar dos punteros en una expresión relacional. Generalmente la comparación de punteros se utiliza cuando dos o más punteros apuntan a un objeto común.

if(p<q) printf (“p apunta a menor memoria que q”);

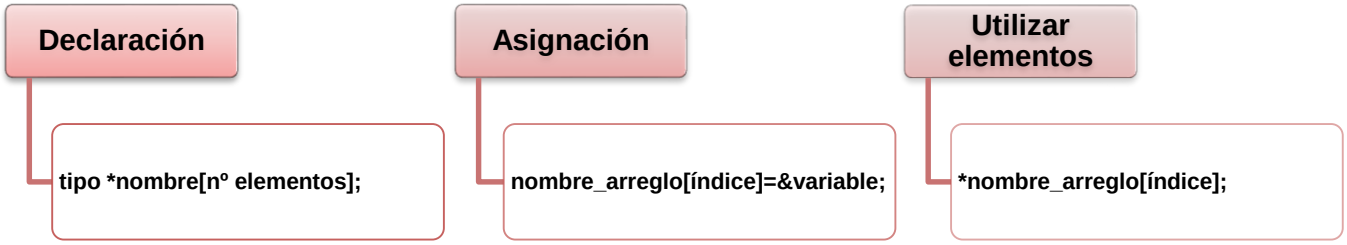
6.2. Apuntadores y arreglos

Existe una relación estrecha entre los punteros y los arreglos. En C, un nombre de un arreglo es un índice a la dirección de comienzo del arreglo. En esencia, el nombre de un arreglo es un puntero al arreglo. Considerar lo siguiente:

```
int a[10], x;
int *ap;
ap = &a[0];    /* ap apunta a la dirección de a[0] */
x = *ap;       /* A x se le asigna el contenido de ap (a[0] en este caso) */
*(ap + 1) = 100;  /* Se asigna al segundo elemento de 'a' el valor 100 usando ap*/
```

Como se puede observar en el ejemplo, la sentencia a[t] es idéntica a ap+t. Se debe tener cuidado ya que C no hace una revisión de los límites del arreglo, por lo que se puede ir fácilmente más allá del arreglo en memoria y sobrescribir otras cosas.

Los apuntadores pueden estructurarse en arreglos como cualquier otro tipo de datos. Hay que indicar el tipo y el número de elementos. Su utilización posterior es igual a los arreglos que hemos visto anteriormente, con la diferencia de que se asignan direcciones de memoria.



Ejercicio

El siguiente programa utiliza un arreglo de apuntadores. El usuario debe introducir un número de día, y dependiendo de la selección, el programa el día adecuado.

```
#include <stdio.h>
#include <conio.h>
void dias(int n);
void main(void)
{
   int num;
   printf("Introducir nº de Dia: ");
   scanf("%d",&num);
   dias(num);
   getch();
}
void dias(int n)
{

// En la siguiente línea de código se declara el
// arreglo de apuntadores Nombrado *dia[]

   char *dia[]={"Nº de dia no Valido",
   "Lunes",
   "Martes",
   "Miércoles",
   "Jueves",
   "viernes",
   "Sábado",
   "Domingo"};

// en la siguiente línea se imprime las cadenas de los
// nombres de los días de la semana
// contenidas en el arreglo de apuntadores

   printf("%s",día[n]);
```

```
}
```

6.3. Apuntadores y estructuras

C permite punteros a estructuras, igual que permite punteros a cualquier otro tipo de variables. El uso principal de este tipo de punteros es **pasar estructuras a funciones**. Si tenemos que pasar toda la estructura, el tiempo de ejecución puede hacerse eterno; utilizando punteros sólo se pasa la dirección de la estructura. Los punteros a estructuras se declaran poniendo ***** delante de la etiqueta de la estructura.

SINTAXIS	struct nombre_estructura etiqueta; struct nombre_estructura *nombre_puntero;

Para encontrar la dirección de una etiqueta de estructura, se coloca el operador **&** antes del nombre de la etiqueta. Para acceder a los miembros de una estructura usando un puntero usaremos el operador flecha (**->**).

Ejemplo

El siguiente programa utiliza una estructura con apuntadores para almacenar los datos de un empleado, se utiliza una estructura con tres elementos: nombre, dirección y teléfono, se utiliza además una función para mostrar el nombre del empleado en pantalla.

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

```
typedef struct
{
        char nombre[50];
        char dirección[50];
        long int teléfono;
}empleado
void mostrar_nombre(empleado *x);
        int main(void)
        {
                empleado mi_empleado;
                empleado * ptr;
// sintaxis para modificar un miembro de
// una estructura a través de un apuntador
                ptr = &mi_empleado;
                ptr->teléfono = 5632332;
                strcpy(ptr->nombre, "jonas");
                mostrar_nombre(ptr);
                return 0;
        }
        void mostrar_nombre(empleado *x)
        {
// se debe de usar -> para referenciar a
// los miembros de la estructura
                printf("nombre del empleado x %s \n", x->nombre);
        }
```

La estructura se pasa por referencia:

```
mostrar_nombre(ptr);
```

Cuando se ejecuta la anterior instrucción es como si se "ejecutara" la instrucción:

```
Empleado *x = ptr;
```

6.4. Apuntadores y funciones

Cuando C pasa *por valor* los argumentos a las funciones, el parámetro es modificado dentro de la función; una vez que termina la función, el valor anterior de la variable, permanece inalterado.

Hay muchos casos en que se quiere alterar el argumento pasado a la función y recibir el nuevo valor una vez que la función ha terminado. Para hacer lo anterior, se debe usar una *llamada por referencia*, en C se puede hacer pasando un puntero al argumento. Con esto se provoca que la computadora pase la dirección del argumento a la función.

Utilizando apuntadores se puede realizar una rutina de ordenación independiente del tipo de datos, una manera de lograrlo es usar apuntadores sin tipo (void) para que apunten a los datos en lugar de usar el tipo de datos enteros.

Ejercicio

```
#include <stdio.h>
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};
void bubble(int *p, int N);
int compare(int *m, int *n);
int main(void)
{
```

```
int i;
putchar('\n');
for (i = 0; i < 10; i++)
{
printf("%d ", arr[i]);
}
bubble(arr,10);
putchar('\n');
for (i = 0; i < 10; i++)
{
printf("%d ", arr[i]);
}
return 0;
}

void bubble(int *p, int N)
{
int i, j, t;
for (i = N-1; i >= 0; i--)
{
for (j = 1; j <= i; j++)
{
if (compare(&p[j-1], &p[j]))
{
t = p[j-1];
p[j-1] = p[j];
p[j] = t;
}
}
}
}

int compare(int *m, int *n)
{
return (*m > *n);
}
```

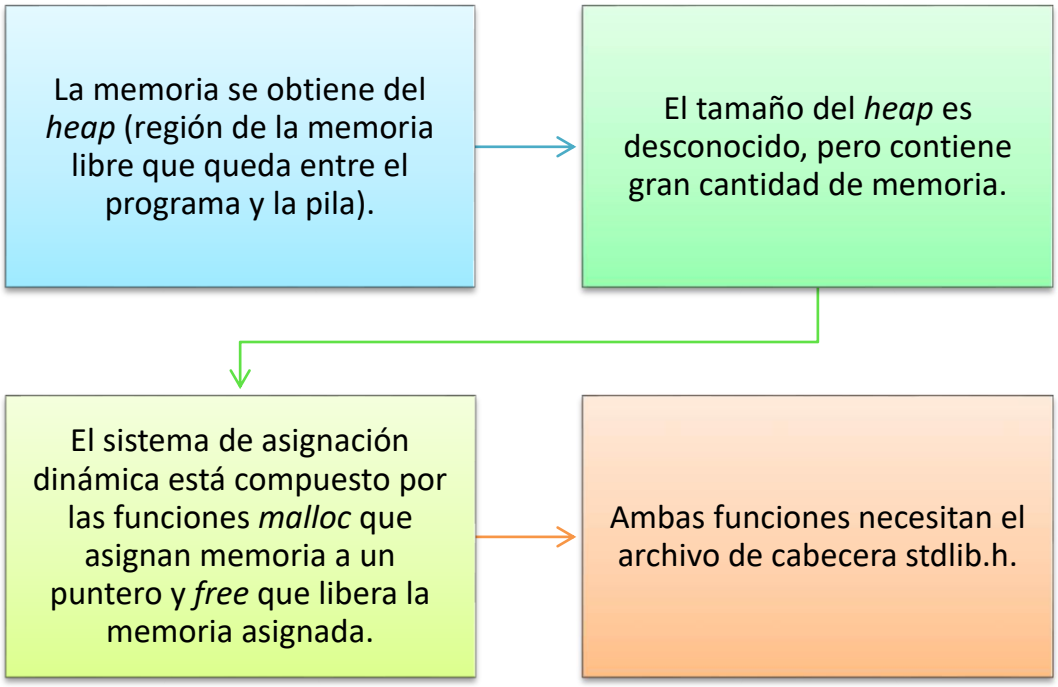
Estamos pasando un apuntador a un entero (o a un arreglo de enteros) a bubble().

Desde dentro de bubble estamos pasando apuntadores a los elementos que queremos comparar del arreglo a nuestra función de comparación, y por supuesto, estamos

desreferenciando estos apuntadores en nuestra función compare() de modo que se haga la comparación real entre elementos. Nuestro siguiente paso será convertir los apuntadores en bubble() a apuntadores sin tipo, de tal modo que la función se vuelva más insensible al tipo de datos por ordenar.

6.5. Manejo dinámico de memoria

La asignación dinámica es la forma en que un programa puede obtener memoria mientras se está ejecutando, debido a que hay ocasiones en que se necesita usar diferente cantidad de memoria.



Al llamar a *malloc*, se asigna un bloque contiguo de almacenamiento al objeto especificado, devolviendo un puntero al comienzo del bloque. La función *free* libera memoria previamente asignada dentro del *heap*, permitiendo que ésta sea reasignada cuando sea necesario.

El argumento pasado a *malloc* es un entero sin signo que representa el número de bytes de almacenamiento requeridos. Si el almacenamiento está disponible, *malloc* devuelve un void *, que se puede transformar en el puntero de tipo deseado. El concepto de punteros void se introdujo en el C ANSI estándar, y significa un puntero de tipo desconocido, o un puntero genérico.

SINTAXIS	puntero=malloc(numero_de_bytes); free(puntero);

El siguiente segmento de código asigna almacenamiento suficiente para 200 valores flota:

```
float *apun_float;
int num_floats = 200;
apun_float = malloc(num_floats * sizeof(flota));
```

La función malloc se ha utilizado para obtener almacenamiento suficiente para 200 por el tamaño actual de un float. Cuando el bloque ya no es necesario, se libera por medio de:

```
free(apun_float);
```

Ejercicio

Después de que el programa define la variable int *block_mem, se llama a la función malloc, para asignar un espacio de memoria suficiente para almacenar una variable de tipo int. Para obtener el tamaño adecuado se usa la función sizeof. Si la asignación es exitosa se manda un mensaje en pantalla, de lo contrario se

indica que la memoria es insuficiente; por último, se libera la memoria ocupada a través de la función free.

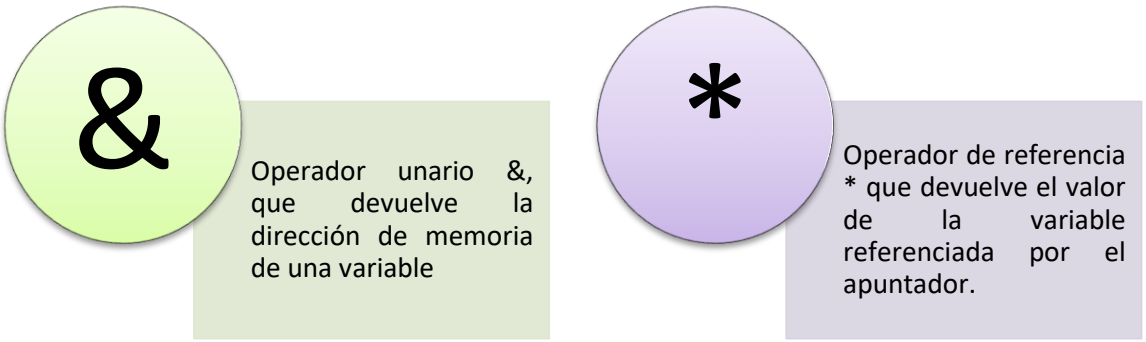
```
#include <stdio.h>
#include <stdlib.h>

#define MAX 256

main()
{
		int *block_mem;
		block_mem=(int *)malloc(MAX * sizeof(int));
		if(block_mem == NULL)
			printf("Memoria insuficiente\n");
		else {
			printf("Memoria asignada\n");
			free(block_mem);
		}

return(0);
}
```

RESUMEN

En esta unidad se estudiaron las características más importantes de los apuntadores en el lenguaje C, así como su relación con los arreglos, estructuras y funciones.

Un apuntador es una variable que contiene una dirección de memoria de otra variable. Los apuntadores, también conocidos como punteros, se utilizan para el almacenamiento de memoria de manera dinámica.

Los apuntadores utilizan dos operadores:



Los apuntadores permiten el uso de las operaciones aritméticas de suma y resta, de modo que cuando se incrementa el puntero, apunta a la posición de memoria del siguiente elemento de su tipo base; cuando se reduce, apunta a la posición del elemento anterior.

Los apuntadores pueden estructurarse en arreglos indicando el tipo y el número de elementos; a los elementos del arreglo se les asignarán direcciones de memoria.

Se pueden utilizar punteros para transferir estructuras a las funciones, pasando únicamente la dirección de la estructura en lugar de cada elemento de ésta.

Los punteros a estructuras se declaran anteponiendo el operador * a la etiqueta de la estructura.

Se usa el operador & para encontrar la dirección de una etiqueta de estructura; para acceder a los miembros de la estructura usando el puntero se utiliza el operador de flecha (**->**).

Para alterar el valor del argumento anterior a la función y recibir el nuevo valor una vez que la función ha terminado, se debe usar una llamada por referencia, pasando un puntero como argumento; con esto se transfiere la dirección del argumento a la función, permitiendo que esta altere el valor almacenado en la dirección del argumento.

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Stroustrup (2002)	5	91-110
Deitel (2004)	7	233-286
	12	421-432

REFERENCIAS BIBLIOGRÁFICAS

SUGERIDA

Brian W.K., y Dennis M. R. (1991) *El lenguaje de programación C.* México: Pearson Educación.

Cairó, O. (2003). *Metodología de la Programación: Algoritomos, diagramas de flujo y programas*. (2ª ed.) México: Alfaomega.

Deitel, H.M. y Deitel, P.J. (2004). *Cómo programar en C/C++ y Java*. (4ª ed.) México: Prentice Hall.

Joyanes, L. (2003). *Fundamentos de Programación: algoritmos y estrutura de datos.* (3ª ed.) Madrid: Pearson / Prentice Hall.

Joyanes, L. (2005). *Algoritmos, estructuras de datos y objetos* (2ª. ed.) Madrid. McGraw-Hill.

Kenneth, L. (2004). *Lenguajes de programación Principios y práctica* (2ª. ed.) México: International Thompson.

Stroustrup, B. (2002). *El lenguaje de programación C++.* (Edición especial) Madrid: Addison Wesley.

BÁSICA

Domínguez, E. D. (2014). *Programación estructurada: raptor y lenguaje C*. México: Alfaomega.

López, L. (2011). *Programación estructurada y orientada a objetos*: un enfoque algorítmico. México: Alfaomega.

Malik, D. S. & León Cárdenas, J. (2013). *Programación Java: del análisis de problemas al diseño de programas.* México: Cengage Learning.

Malik, D. S., & Mercado, E. C., (2013). *Programación C++: del análisis de problemas al diseño de programas.* México: Cengage Learning.

Márquez, T. G., Osorio, S., & Olvera, E. N. (2011). *Introducción a la programación estructurada en C*. México: Pearson.

Martín, C., Urquía, A., & Rubio, M. Á. (2011). *Lenguajes de programación.* España: Universidad Nacional de Educación a Distancia.

Martínez, R., García y Beltrán, A., Tapia & Álamo, F. J. (2014). *Programación en C.*: ejercicios. España: Dextra editorial/ Universidad politécnica de Madrid, Sección de publicaciones de la escuela técnica superior de ingenieros industriales.

Moreno, J. C. (2014). *Programación en lenguajes estructurados*. España: Ra-Ma.

Sierra, A., & Alfonseca, M. (2014). *Programación en C/C++.* España: Anaya Multimedia.

COMPLEMENTARIA

Cheverri, J. A., & Orrego, G. A. (2012). *Programación. Teoría y aplicaciones*. Colombia: Ediciones de la U.

Juganaru, M. (2012). *Introducción a la programación.* México: Patria.

Kolling, M., (2011). *Introducción a la programación con Greenfoot: programación orientada a objetos en JavaTM con juegos y simulaciones*. España: Pearson

Menchén, A. (2010). *Diseño de programas*. México: Alfaomega.

Moreno, J. C. (2013). *Programación*. Colombia: Ediciones de la U.

SITIOS ELECTRÓNICOS

Sitio	Descripción
http://www.data-2013.cl/DOCS/INFORMATICA/PROGRC/cap-c9.html	Biblioteca de funciones. Funciones de C y C++

Plan 2012

2016

actualizado