

**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**  
FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

DIVISIÓN SISTEMA UNIVERSIDAD ABIERTA Y  
EDUCACIÓN A DISTANCIA

L I C E N C I A T U R A en  
**INFORMÁTICA**

**APUNTES DIGITALES**  
**PLAN 2012**



**SUAYED** UNA OPCIÓN  
PARA TI



# INTRODUCCIÓN A LA PROGRAMACIÓN

**Plan 2012**

<b>Clave:</b>		<b>Créditos: 8</b>
<b>Licenciatura: INFORMÁTICA</b>		<b>Semestre: 2º</b>
<b>Área: Desarrollo de sistemas</b>		<b>Horas asesoría: 64</b>
<b>Requisitos:</b>		<b>Horas por semana: 4</b>
<b>Tipo de asignatura:</b>	<b>Obligatoria ( X )</b>	<b>Optativa ( )</b>

**AUTOR**

ESPARTACO DAVID KANAGUSICO HERNANDEZ

**ADAPTACIÓN EN LÍNEA**

ESPARTACO DAVID KANAGUSICO HERNANDEZ

**ACTUALIZACIÓN AL PLAN DE ESTUDIOS 2012**

GILBERTO MANZANO PEÑALOZA



SUAYED  
Sistema Universitario  
Abierto y a Distancia

# INTRODUCCIÓN AL MATERIAL DE ESTUDIO

Las modalidades abierta y a distancia (SUAYED) son alternativas que pretenden responder a la demanda creciente de educación superior, sobre todo, de quienes no pueden estudiar en un sistema presencial. Actualmente, señala Sandra Rocha (2006):

con la incorporación de las nuevas tecnologías de información y comunicación a los sistemas abierto y a distancia, se empieza a fortalecer y consolidar el paradigma educativo de éstas, centrado en el estudiante y su aprendizaje autónomo, para que tenga lugar el diálogo educativo que establece de manera semipresencial (modalidad abierta) o vía Internet (modalidad a distancia) con su asesor y condiscípulos, apoyándose en materiales preparados ex profeso.

Un rasgo fundamental de la educación abierta y a distancia es que no exige presencia diaria. El estudiante SUAYED aprende y organiza sus actividades escolares de acuerdo con su ritmo y necesidades; y suele hacerlo en momentos adicionales a su jornada laboral, por lo que requiere flexibilidad de espacios y tiempos. En consecuencia, debe contar con las habilidades siguientes.

- Saber estudiar, organizando sus metas educativas de manera realista según su disponibilidad de tiempo, y estableciendo una secuencia de objetivos parciales a corto, mediano y largo plazos.



- Mantener la motivación y superar las dificultades inherentes a la licenciatura.
- Asumir su nuevo papel de estudiante y compaginarlo con otros roles familiares o laborales.
- Afrontar los cambios que puedan producirse como consecuencia de las modificaciones de sus actitudes y valores, en la medida que se adentre en las situaciones y oportunidades propias de su nueva situación de estudiante.
- Desarrollar estrategias de aprendizaje independientes para que pueda controlar sus avances.
- Ser autodidacta. Aunque apoyado en asesorías, su aprendizaje es individual y requiere dedicación y estudio. Acompañado en todo momento por su asesor, debe organizar y construir su aprendizaje.
- Administrar el tiempo y distribuirlo adecuadamente entre las tareas cotidianas y el estudio.
- Tener disciplina, perseverancia y orden.
- Ser capaz de tomar decisiones y establecer metas y objetivos.
- Mostrar interés real por la disciplina que se estudia, estar motivado para alcanzar las metas y mantener una actitud dinámica y crítica, pero abierta y flexible.
- Aplicar diversas técnicas de estudio. Atender la retroalimentación del asesor; cultivar al máximo el hábito de lectura; elaborar resúmenes, mapas conceptuales, cuestionarios, cuadros sinópticos, etcétera; presentar trabajos escritos de calidad en contenido, análisis y reflexión; hacer guías de estudio; preparar exámenes; y aprovechar los diversos recursos de la modalidad.

Además de lo anterior, un estudiante de la modalidad a distancia debe dominar las herramientas tecnológicas. Conocer sus bases y metodología;



tener habilidad en la búsqueda de información en bibliotecas virtuales; y manejar el sistema operativo Windows, paquetería, correo electrónico, foros de discusión, chats, blogs, wikis, etcétera.

También se cuenta con materiales didácticos como éste elaborados para el SUAYED, que son la base del estudio independiente. En específico, este documento electrónico ha sido preparado por docentes de la Facultad para cada una de las asignaturas, con bibliografía adicional que te permitirá consultar las fuentes de información originales. El recurso comprende referencias básicas sobre los temas y subtemas de cada unidad de la materia, y te introduce en su aprendizaje, de lo concreto a lo abstracto y de lo sencillo a lo complejo, por medio de ejemplos, ejercicios y casos, u otras actividades que te posibilitarán aplicarlos y vincularlos con la realidad laboral. Es decir, te induce al “saber teórico” y al “saber hacer” de la asignatura, y te encauza a encontrar respuestas a preguntas reflexivas que te formules acerca de los contenidos, su relación con otras disciplinas, utilidad y aplicación en el trabajo. Finalmente, el material te da información suficiente para autoevaluarte sobre el conocimiento básico de la asignatura, motivarte a profundizarlo, ampliarlo con otras fuentes bibliográficas y prepararte adecuadamente para tus exámenes. Su estructura presenta los siguientes apartados.

1. *Información general de la asignatura.* Incluye elementos introductorios como portada, identificación del material, colaboradores, datos oficiales de la asignatura, orientaciones para el estudio, contenido y programa oficial de la asignatura, esquema general de contenido, introducción general a la asignatura y objetivo general.
2. *Desarrollo de cada unidad didáctica.* Cada unidad está conformada por los siguientes elementos:
  - Introducción a la unidad.



Objetivo específico de la unidad.

Contenidos.

Actividades de aprendizaje y/o evaluación. Tienen como propósito contribuir en el proceso enseñanza-aprendizaje facilitando el afianzamiento de los contenidos esenciales. Una función importante de estas actividades es la retroalimentación: el asesor no se limita a valorar el trabajo realizado, sino que además añade comentarios, explicaciones y orientación.

Ejercicios y cuestionarios complementarios o de reforzamiento. Su finalidad es consolidar el aprendizaje del estudiante.

Ejercicios de autoevaluación. Al término de cada unidad hay ejercicios de autoevaluación cuya utilidad, al igual que las actividades de aprendizaje, es afianzar los contenidos principales. También le permiten al estudiante calificarse él mismo cotejando su resultado con las respuestas que vienen al final, y así podrá valorar si ya aprendió lo suficiente para presentar el examen correspondiente. Para que la autoevaluación cumpla su objeto, es importante no adelantarse a revisar las respuestas antes de realizar la autoevaluación; y no reducir su resolución a una mera actividad mental, sino que debe registrarse por escrito, labor que facilita aún más el aprendizaje. Por último, la diferencia entre las actividades de autoevaluación y las de aprendizaje es que éstas, como son corregidas por el asesor, fomentan la creatividad, reflexión y valoración crítica, ya que suponen mayor elaboración y conllevan respuestas abiertas.

3. *Resumen* por unidad.

4. *Glosario* de términos.



5. *Fuentes* de consulta básica y complementaria. Mesografía, bibliografía, hemerografía, sitios web, entre otros, considerados tanto en el programa oficial de la asignatura como los sugeridos por los profesores.

Esperamos que este material cumpla con su cometido, te apoye y oriente en el avance de tu aprendizaje.



## Recomendaciones (orientación para el estudio independiente):

- Lee cuidadosamente la introducción a la asignatura, en ella se explica la importancia del curso.
- Revisa detenidamente los objetivos de aprendizaje (general y específico por unidad), en donde se te indican los conocimientos y habilidades que deberás adquirir al finalizar el curso.
- Estudia cada tema siguiendo los contenidos y lecturas sugeridos por tu asesor, y desarrolla las actividades de aprendizaje. Así podrás aplicar la teoría y ejercitarás tu capacidad crítica, reflexiva y analítica.
- Al iniciar la lectura de los temas, identifica las ideas, conceptos, argumentos, hechos y conclusiones, esto facilitará la comprensión de los contenidos y la realización de las actividades de aprendizaje.
- Lee de manera atenta los textos y mantén una actitud activa y de diálogo respecto a su contenido. Elabora una síntesis que te ayude a fijar los conceptos esenciales de lo que vas aprendiendo.
- Debido a que la educación abierta y a distancia está sustentada en un principio de autoenseñanza (autodisciplina), es recomendable diseñar desde el inicio un plan de trabajo para puntualizar tiempos, ritmos, horarios, alcance y avance de cada asignatura, y recursos.



SUAVED  
SERVICIO DE  
ASesoría

- Escribe tus dudas, comentarios u observaciones para aclararlas en la asesoría presencial o a distancia (foro, chat, correo electrónico, etcétera).
- Consulta al asesor sobre cualquier interrogante por mínima que sea.
- Revisa detenidamente el plan de trabajo elaborado por tu asesor y sigue las indicaciones del mismo.



SUAYED

## Otras sugerencias de apoyo

- Trata de compartir tus experiencias y comentarios sobre la asignatura con tus compañeros, a fin de formar grupos de estudio presenciales o a distancia (comunidades virtuales de aprendizaje, a través de foros de discusión y correo electrónico, etcétera), y puedan apoyarse entre sí.
- Programa un horario propicio para estudiar, en el que te encuentres menos cansado, ello facilitará tu aprendizaje.
- Dispón de periodos extensos para al estudio, con tiempos breves de descanso por lo menos entre cada hora si lo consideras necesario.
- Busca espacios adecuados donde puedas concentrarte y aprovechar al máximo el tiempo de estudio.



SUAYED

# TEMARIO DETALLADO

(Horas sugeridas 64)

	Horas
1. Introducción a la programación	4
2. Tipos de datos elementales (variables, constantes, declaraciones, 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
<b>TOTAL</b>	<b>66</b>



# 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.

En la *primera unidad* (introducción a la programación) se mencionan conceptos básicos de programación. 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. En la *tercera unidad* (control de flujo) se analiza la utilización de la estructura secuencial, condicional y repetitiva. En la *cuarta unidad* (funciones) se analiza la función, y su utilidad para la realización de tareas específicas dentro de un programa. 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. En la *sexta unidad* (manejo de apuntadores) se utilizan los apuntadores para la realización de programas que utilizan memoria dinámica.

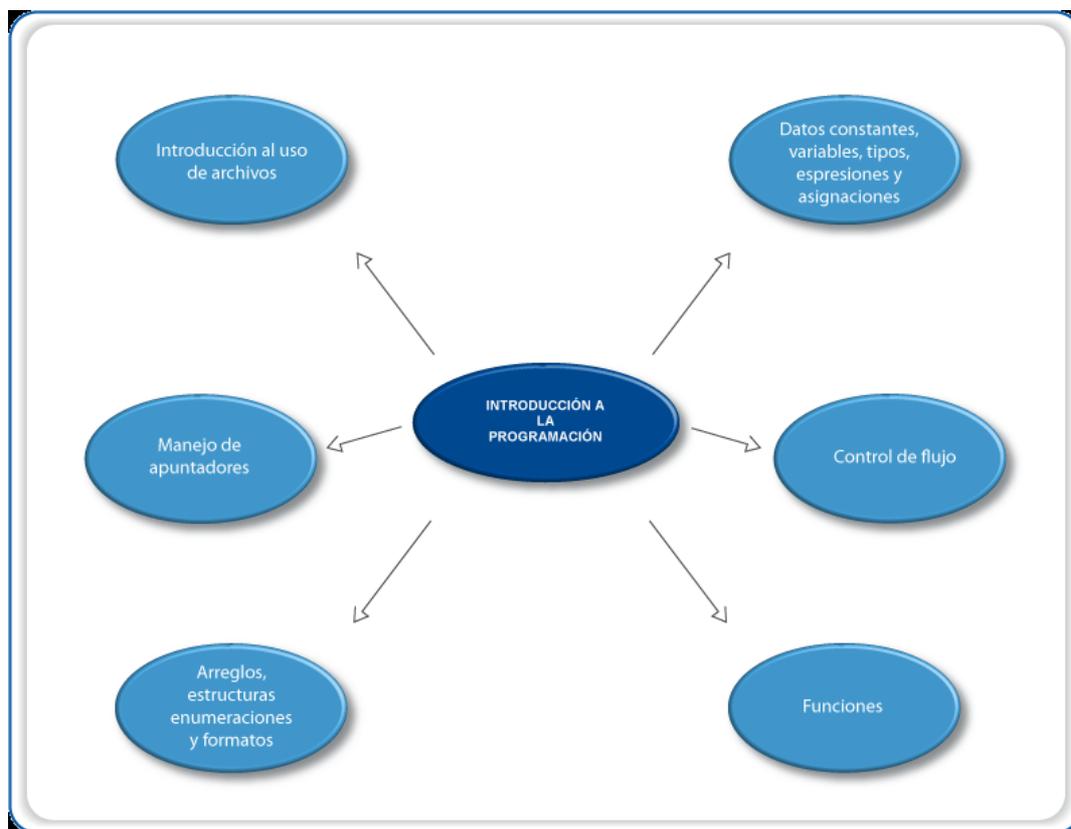


## OBJETIVO GENERAL

Al finalizar el curso, el alumno será capaz de implementar algoritmos en un lenguaje de programación.



## ESTRUCTURA CONCEPTUAL





SUAYED  
UNA OPCIÓN  
PARA TI

Licenciatura: Informática

## UNIDAD 1

# INTRODUCCIÓN A LA PROGRAMACIÓN

## APUNTES DIGITALES PLAN 2012



SUAYED  
UNA OPCIÓN  
PARA TI



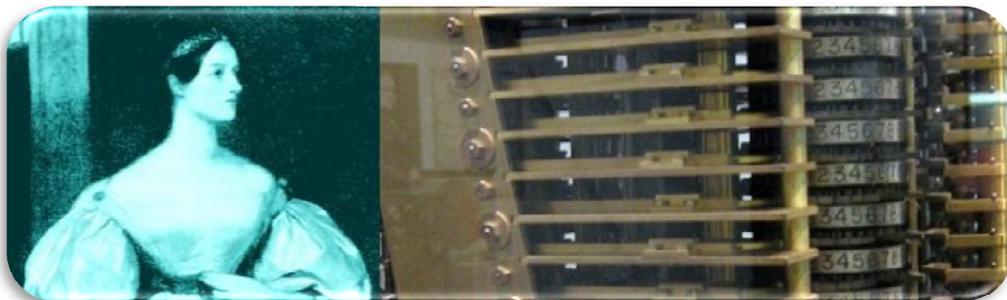
## OBJETIVO ESPECÍFICO

Al terminar la unidad, el alumno será capaz establecer la diferencia entre los paradigmas de programación e identificar los lenguajes de acuerdo con su nivel y sus principales características.

## 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.





Podemos definir a un lenguaje de programación como una herramienta que permite desarrollar programas para computadora. Dichos lenguajes permiten expresar las instrucciones que el programador desea se ejecuten en la computadora.

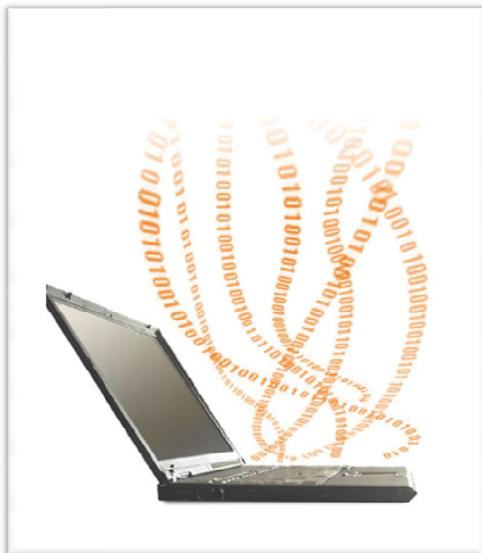
La función de los lenguajes de programación es realizar programas que permitan la resolución de problemas.

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**.

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:

- El concepto de lenguaje de programación.
- El paradigma estructurado.
- El paradigma orientado a objetos.
- Los lenguajes de alto y bajo nivel.
- Intérpretes y compiladores.
- Las fases de la compilación.



SUAYED  
Sistema Universitario  
Autónomo de Uruguay

## LO QUE SÉ

El siguiente cuestionario te permitirá iniciar el aprendizaje de esta unidad con tus conocimientos previos. Cabe destacar que las respuestas de este cuestionario no influyen en tu evaluación.

Con tus propias palabras define:

1. ¿Qué es un lenguaje?
2. ¿Qué es una computadora?
3. ¿Qué entiendes por lenguaje de computadora?
4. Menciona 5 lenguajes de programación que conozcas.



SUAYED  
SISTEMAS DE  
AYUDA  
AL  
ESTUDIO

# TEMARIO DETALLADO

**(4 horas)**

- 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



# 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.

```
010000000001010001101100000010010110001
110001011101000100011111111110100000100
00101001011000011010111011010110110010001
0110110000010101100100010000111000100111
0100110010110100110110100111101111011110
0011010001000100010001000100010001000100
1001001100010001000100010001000100010001
10001001int main()
01010001{
111001100 printf("Hello World");
01000001 return 42;
0011010001000100010001000100010001000100
1001001101111010111011110000001010001110
1000100100010101100100111011101000101111
01010001110011010101110001010100011000
11100110000011011111010100111110001100
0100000111111010100100100110101110111
```



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.

Su sintaxis **es compacta**, debido a que maneja pocas funciones y palabras reservadas (aquellas palabras que son instrucciones o comandos propios del lenguaje de programación), comparado con otros lenguajes, como lo es Java.

Además **es portable**, debido a que puede ser utilizado en varios sistemas operativos y hardware.



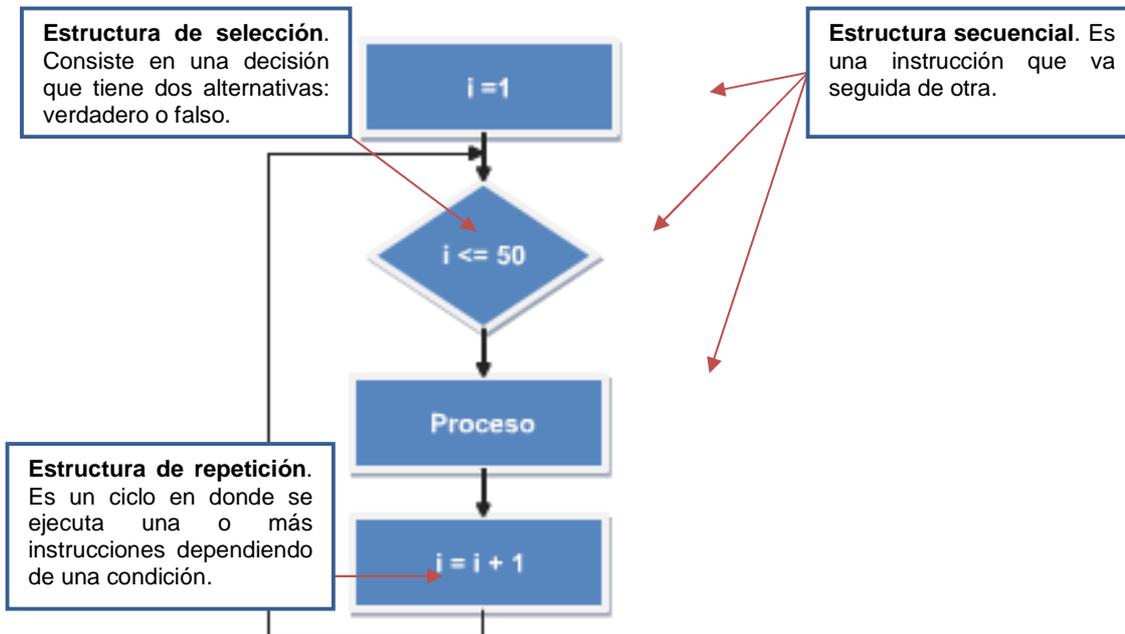
## 1.2. Paradigmas de programación

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.



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*

Ejercicio

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



```
main()  
{  
    printf("Hola mundo");  
    return(0);  
}
```

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

La **primer 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:



- Se invoca a la librería `stdio.h` para poder utilizar las funciones *printf* y *scanf*; la primera para mostrar mensajes en pantalla y la segunda para leer valores ingresados por el usuario.
- En la sentencia `char s[100]` se realiza la declaración de la variable “s” que es tipo carácter con la capacidad de almacenar hasta 100 letras, números o signos de puntuación.
- La instrucción `void main (void)` indica que la función no va recibir ni a devolver valor alguno, por el empleo de la palabra reservada *void*.
- Con la sentencia `scanf(“%s”, s)` se indica que se va a leer una cadena de caracteres y que ésta se va a almacenar en la variable “s”.
- Y la sentencia `printf` va a imprimir el texto “La cadena es: ” seguida del valor que almacena la variable “s” que este caso es el mensaje ingresado por el usuario y posteriormente realizar un salto de línea (`\n`).

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 especifica 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 álgebra, 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 (el primero se refiere a los valores que ocupa una función y el segundo a los valores que resultan). Una definición de función específica



el dominio y rango, de manera implícita o explícita, junto con una expresión que describe la correspondencia.

Las funciones son aplicadas a un elemento del dominio y devuelven uno del rango.

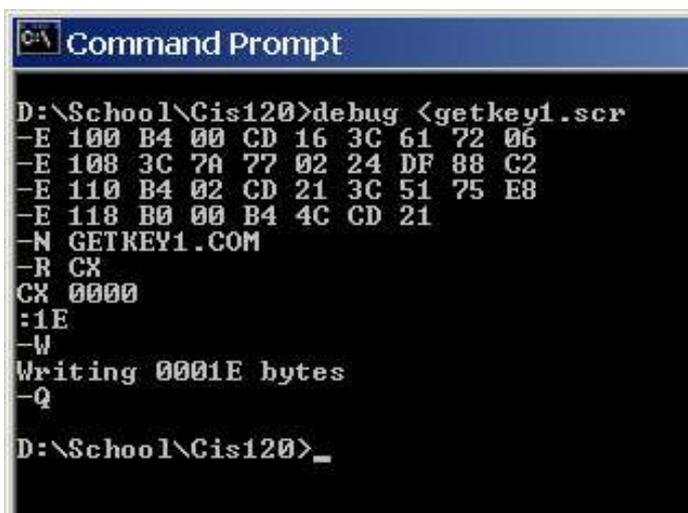
Las funciones matemáticas no producen efectos laterales. Dado un mismo conjunto de argumentos, una función matemática producirá siempre el mismo resultado; por ejemplo, de una función que arroje el mayor de tres números dados, siempre se va a esperar este resultado, independientemente de los valores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

El lenguaje de máquina es el sistema de códigos directamente interpretable por un circuito microprogramable, como el microprocesador de una computadora. 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.



```
C:\ Command Prompt
D:\School\Cis120>debug <getkey1.scr
-E 100 B4 00 CD 16 3C 61 72 06
-E 108 3C 7A 77 02 24 DF 88 C2
-E 110 B4 02 CD 21 3C 51 75 E8
-E 118 B0 00 B4 4C CD 21
-N GETKEY1.COM
-R CX
CX 0000
:1E
-W
Writing 0001E bytes
-Q
D:\School\Cis120>_
```



## 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.

Un lenguaje de programación de bajo nivel es el que proporciona poca o ninguna abstracción del microprocesador de una computadora. Consecuentemente, es fácil su traslado 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.



```
Codigo Maquina
Dump of assembler code for function no_printa:
0x401270 <no_printa>: push  %ebp
0x401271 <no_printa+1>: mov   %esp,%ebp
(*) 0x401273 <no_printa+3>: sub  $0x4,%esp
0x401276 <no_printa+6>: movl $0x2,0xffffffff(%ebp)
0x40127d <no_printa+13>: lea  0xffffffff(%ebp),%eax
0x401280 <no_printa+16>: incl (%eax)
0x401282 <no_printa+18>: leave
0x401283 <no_printa+19>: ret
End of assembler dump.
```



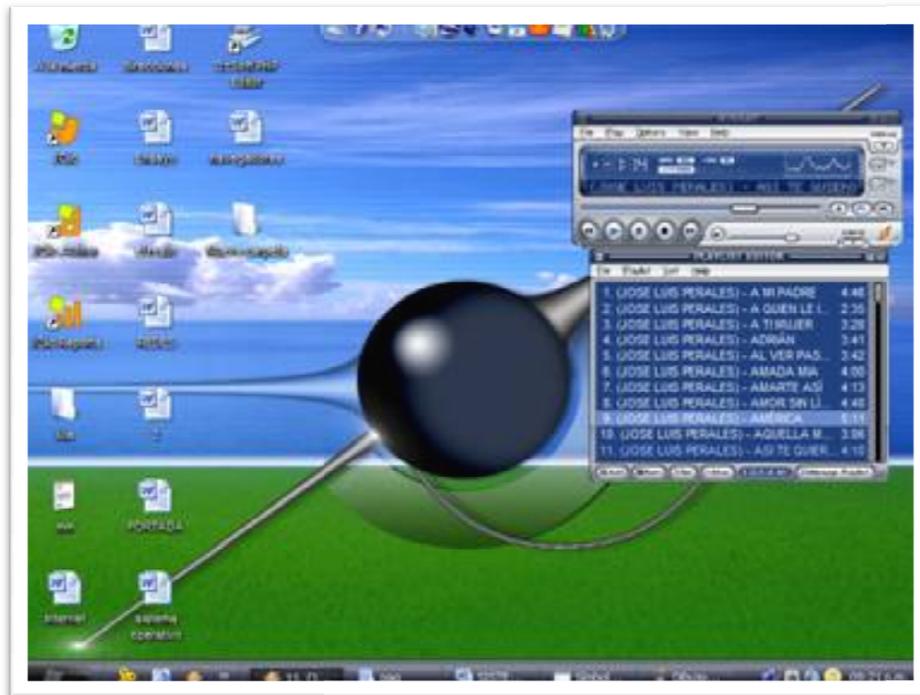
## 1.5. Lenguajes de alto nivel

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 a 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.



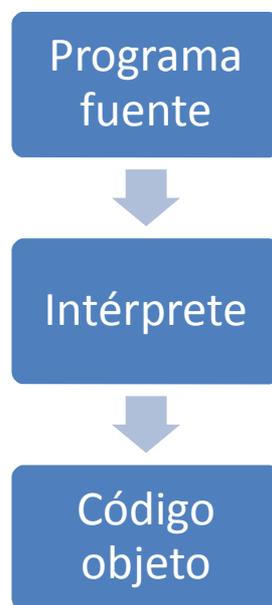
* Perl	Lenguaje práctico para la extracción e informe. Es un lenguaje de programación diseñado por Larry Wall creado en 1987. Perl toma características de C, del lenguaje interpretado shell sh, AWK, sed, Lisp y, en un grado inferior, muchos otros lenguajes de programación.
* PHP	Es un lenguaje de programación 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 última versión estable del lenguaje es actualmente (septiembre 2006) la 3.2.1 (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.



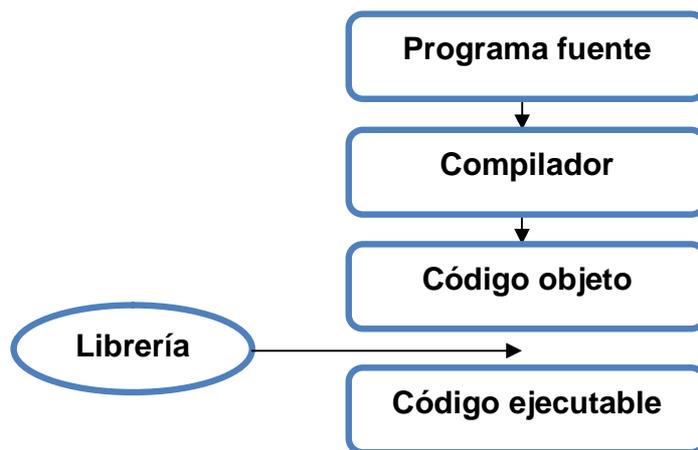
## 1.8. Fases de la compilación

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:

$$\langle \text{símbolo} \rangle ::= \langle \text{expresión con símbolos} \rangle$$

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

$$\begin{aligned} \langle \text{datos trabajador} \rangle ::= & \langle \text{nombre del trabajador} \rangle \text{“,”} \langle \text{edad} \rangle \text{“,”} \langle \text{sexo} \rangle \text{“,”} \\ & \langle \text{salario diario} \rangle \end{aligned}$$

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.
- **Palabras reservadas.** Son palabras clave de un lenguaje, como *for* o *do*.



- **Símbolos especiales.** Son símbolos de puntuación, como \*, >= o.

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 DE LA UNIDAD

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.



SUAYED

## GLOSARIO DE LA UNIDAD

### **Algoritmo**

Conjunto de pasos para resolver un problema.

### **Clase**

En la programación orientada a objetos, una clase es un conjunto de objetos que comparten características comunes.

### **Código fuente**

También denominado fuente o texto fuente. Es el texto que contiene las instrucciones del programa, escritas en el lenguaje de programación. Se trata de un archivo de texto legible que se puede copiar, modificar e imprimir sin dificultad.

El concepto contrario es el código objeto, que se deriva del código fuente y está diseñado para ser legible sólo por la máquina.

### **Código máquina**

El lenguaje máquina es el único lenguaje que puede ejecutar una computadora. El lenguaje de máquina es un código que es interpretado directamente por el microprocesador.

El lenguaje está compuesto por un conjunto de instrucciones ejecutadas en secuencia (con eventuales cambios de flujo causados por el propio



programa o eventos externos) que representan acciones que la máquina podrá tomar.

Un lenguaje máquina es específico de cada arquitectura de computadora. Todo código fuente en última instancia debe llevarse a un lenguaje máquina mediante el proceso de compilación o interpretación para que la computadora pueda ejecutarlo.

### **Código objeto**

En programación, el código objeto es el resultado de la compilación del código fuente. Puede ser en lenguaje máquina o bytecode, y puede distribuirse en varios archivos que corresponden a cada código fuente compilado. Luego un enlazador (linker) se encarga de juntar todos los archivos de código fuente para obtener el programa ejecutable.

### **Compilador**

Programa informático que permite transformar un código fuente a su equivalente en código ejecutable.

### **Diseño modular**

Propiedad de la programación estructurada que permite dividir un programa complejo en problemas más pequeños y fáciles de resolver.

### **Función**

Término con que se describe a una porción del código de un programa, que realiza una tarea específica.

### **Intérprete**

Programa que transforma un código fuente a código objeto.



## **Lenguaje ensamblador**

El lenguaje ensamblador es un lenguaje de programación de bajo nivel, y es la representación más cercana al código máquina (1 y 0).

**Linker** (*Linker*, palabra inglesa que suele traducirse como enlazador)

En programación, un enlazador es un módulo o programa que junta los ficheros de código objetos (generados en la primer parte de la compilación), la información de todos los recursos necesarios (biblioteca), elimina los recursos que no se necesitan y enlaza el código objeto con sus bibliotecas. Finalmente produce el fichero ejecutable o una biblioteca. Existen programas que se enlazan dinámicamente, esto significa que este proceso se hace en el momento que se carga el programa.

## **Objeto**

Un objeto se define como una unidad que realiza las tareas de un programa. También se define como la instancia o caso especial de una clase. || Un objeto es una unidad de código compuesto de variables y métodos relacionados.

## **Paradigma**

Un paradigma es un modelo o patrón por seguir.

## **Parámetros de una función**

Conjunto de variables que utiliza una función. Valores de cualquier tipo que se pasan a la función para que esta pueda realizar su tarea.



### **Programación estructurada**

La programación estructurada divide un problema complejo, en problemas más pequeños para poder resolverlo de una manera más fácil.

### **Programación orientada a objetos**

Es un paradigma de programación donde los programas se organizan con colecciones cooperativas de objetos, un objeto es una representación de una entidad del mundo real, como un individuo, un automóvil, una casa, etcétera.

### **Token**

En programación, un elemento individual en un lenguaje de programación. Es un bloque de texto categorizado. Por ejemplo una marca de puntuación, un operador, un identificador, un número, etc.



## ACTIVIDADES DE APRENDIZAJE

### ACTIVIDAD 1

Busca la historia del lenguaje de programación C.

### ACTIVIDAD 2

Investiga acerca de las cuatro primeras generaciones de los lenguajes de programación.

### ACTIVIDAD 3

Realiza un algoritmo que sume tres números. Para mayor información de cómo realizar un algoritmo, visita el sitio <http://lawebdelprogramador.com>

### ACTIVIDAD 4

Modifica este programa para que sume tres números.

```
# include <stdio.h>
main()
{
    int c=0;
    int a=2;
```



```
int b=3;
a= a+b;
printf("%d",c);
return(0);
}
```

### ACTIVIDAD 5

Busca sobre los conceptos: objeto, clase y herencia. Apóyate en el documento *El paradigma orientado a objetos* ([ANEXO 2](#)). No olvides colocar tus fuentes bibliográficas.

### ACTIVIDAD 6

Elabora un cuadro comparativo de la programación orientada a objetos y la programación imperativa.

### ACTIVIDAD 7

Realiza una búsqueda sobre:

- Lenguaje ensamblador
- Lenguaje de bajo nivel
- Lenguaje de alto nivel

Identifica sus iniciadores, conceptos, principales características, diferencias entre los lenguajes, ventajas desventajas, estructura, y proporciona 2 ejemplos de cada uno.



**ACTIVIDAD 8**

Elabora un programa en lenguaje ensamblador que sume dos números.

**ACTIVIDAD 9**

Investiga y enlista 5 lenguajes de programación que sean de alto nivel.

**ACTIVIDAD 10**

Investiga 5 lenguajes de programación que utilicen un intérprete y 5 que empleen compiladores.

**ACTIVIDAD 11**

Investiga y elabora un cuadro comparativo de las diferencias entre el intérprete y el compilador.

**ACTIVIDAD 12**

Elabora un programa en C e identifica los pasos que realiza el compilador para generar el programa ejecutable.



# CUESTIONARIO DE REFORZAMIENTO

*Responde las siguientes preguntas.*

1. Señala qué es un lenguaje de programación.
2. ¿Qué es el código fuente?
3. ¿Qué es el código objeto?
4. ¿Qué es el código ejecutable?
5. ¿En qué nivel se clasifica al lenguaje C y por qué?
6. ¿Qué es un algoritmo?
7. ¿Qué es un programa?
8. ¿Qué es un compilador?
9. ¿Qué es un intérprete?
10. ¿Qué es un objeto?



## EXAMENES DE AUTOEVALUACIÓN

I. Selecciona si las aseveraciones son verdaderas o falsas.

	Verdadera	Falsa
1. La programación estructurada utiliza un diseño modular.	( )	( )
2. C es un lenguaje estructurado.	( )	( )
3. Un ejemplo de una estructura de control es la iteración:	( )	( )
4. <i>Printf</i> muestra un mensaje en pantalla.	( )	( )
5. <i>Scanf</i> muestra un mensaje en pantalla.	( )	( )
6. La programación estructurada utiliza objetos	( )	( )
7. C es un lenguaje orientado a objetos.	( )	( )
8. Un ejemplo de una estructura de control es la iteración	( )	( )
9. Una función es sinónimo de una clase.	( )	( )
10. <i>#include</i> se utiliza para acceder a una librería de funciones.	( )	( )
11. Todos los leguajes de programación usan compatibilidades	( )	( )
12. C ++ es un lenguaje orientado a objetos.	( )	( )
13. La programación orientada a objetos divide un problema complejo en problemas más sencillos	( )	( )
14. El lenguaje PHP utiliza un compilador.	( )	( )
15. El lenguaje C es un ejemplo de un lenguaje de	( )	( )



bajo nivel.



II. Relaciona las columnas. Escribe la letra correcta en el espacio que le corresponde.

- |   |  |
|---|--|
| <p>( ) 1. Este lenguaje se refiere a un sistema de códigos directamente interpretable por un circuito micro programable.</p>                                      | <p><b>A</b> PHP</p>                      |
| <p>( ) 2. Consiste en una cadena de instrucciones de lenguaje de máquina.</p>   | <p><b>B</b> Ensamblador</p>              |
| <p>( ) 3. Este lenguaje proporciona poca o ninguna abstracción del microprocesador de una computadora.</p>  | <p><b>C</b> Lenguaje de máquina</p>      |
| <p>( ) 4. Se refiere a un tipo de programa informático que se encarga de traducir un archivo fuente escrito en un archivo objeto que contiene código máquina.</p> | <p><b>D</b> Lenguaje de bajo nivel</p>   |
| <p>( ) 5. Se caracteriza por expresar los algoritmos de una manera adecuada a la capacidad cognitiva humana.</p>  | <p><b>E</b> Programa de computadoras</p> |
| <p>( ) 6. Es ejemplo de un lenguaje de alto nivel.</p>  | <p><b>F</b> Lenguaje de alto nivel</p>   |



*III. Selecciona la opción que consideres correcta:*

1. El código fuente es un:
  - a) código de computadora.
  - b) diagrama de flujo.
  - c) paradigma.
  - d) conjunto de caracteres entendibles por un ser humano.
  
2. El código objeto es:
  - a) un código entendible por la computadora.
  - b) sinónimo de un intérprete.
  - c) sinónimo de un compilador.
  - d) un código entendible por un ser humano.
  
3. El código ejecutable:
  - a) se obtiene usando un linker.
  - b) se obtiene usando un intérprete.
  - c) es un sinónimo de paradigma.
  - d) es un sinónimo de objeto.
  
4. Una clase es:
  - a) sinónimo de objeto.
  - b) sinónimo de herencia.
  - c) el conjunto de funciones con características similares.
  - d) el conjunto de objetos con características similares.



5. La función tiene su equivalente, en la programación orientada a objetos, en el concepto de:
- a) herencia.
  - b) clase.
  - c) método.
  - d) objeto.
6. Un intérprete lee línea por línea:
- a) el código fuente
  - b) el código objeto.
  - c) el código ejecutable.
  - d) las librerías.
7. Ejemplo de una librería:
- a) scanf
  - b) void
  - c) stdio.h
  - d) Java.
8. Python utiliza un:
- a) enlazador.
  - b) compilador y un intérprete.
  - c) compilador.
  - d) intérprete.



9. En un programa, un linker:
- a) agrega librerías.
  - b) genera un código objeto.
  - c) genera un código fuente.
  - d) es sinónimo de clase.



## LO QUE APRENDÍ

Escribe el código fuente de un programa que obtenga el promedio de tres números, empleando cada uno de los paradigmas de programación: imperativo, orientado a objetos y funcional.



# MESOGRAFÍA

## Bibliografía sugerida

Autor	Capítulo	Páginas
Cairó (2003)	2	250-252
Joyanes (2003)	2	47-74
Kenneth (2004)	1	1-30
	4	69-112
	5	113-172

## Bibliografía básica

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

Ceballos, Francisco Javier. (1997). *Lenguaje C*. México: Alfaomega.

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



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

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

## Bibliografía complementaria

García, L.; Cuadrado, J.; De Amescua, A. y Velasco, M. (2004) *Construcción lógica de programas, Teoría y problemas resueltos*. México: Alfa Omega / Ra-Ma.

Joyanes Aguilar, Luis. (2005). *Programación en C++: Algoritmos, Estructuras de Datos y objetos*. (2ª ed.) Madrid: McGraw-Hill.

López, Leobardo. (2004). *Programación estructurada en turbo pascal 7*. México: Alfa Omega.

----- (2004a). *Programación estructurada, un enfoque algorítmico*. (2ª. ed.) México: Alfa Omega.

Sedgewick, Robert. (1995). *Algoritmos en C++*. México: Addison-Wesley Iberoamericana.

Zárate Mejía, E. en (2002). Román Zamitiz, Carlos Alberto. (2002). *Programación Orientada a Objetos con Java*. En temas



especiales de computación, Facultad de Ingeniería,  
México: UNAM, disponible en línea: [http://profesores.fi-  
b.unam.mx/carlos/java/indice.html](http://profesores.fi-b.unam.mx/carlos/java/indice.html), consultado el 12/09/11



## Sitios de internet

(Nota: todos los vínculos de este texto, consultados o recuperados, funcionan al 23/01/12)

Sitio	Descripción
<a href="http://c.conclase.net/">http://c.conclase.net/</a>	Cursos relacionados con C y C++
<a href="http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B">http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B</a>	Libro de programación en C++
<a href="http://es.wikipedia.org/wiki/C%2B%2B">%2B</a>	Artículo sobre el lenguaje C++
<a href="http://www.desarrolloweb.com/articulos/1514.php">http://www.desarrolloweb.com/articulos/1514.php</a>	Nuevas versiones de PHP
<a href="http://java.ciberaula.com/articulo/tecnologia_orientada_objetos/">http://java.ciberaula.com/articulo/tecnologia_orientada_objetos/</a>	Programación orientada a objetos
<a href="http://profesores.fi-b.unam.mx/carlos/java/java_basico3_1.html">http://profesores.fi-b.unam.mx/carlos/java/java_basico3_1.html</a>	Clases y objetos
<a href="http://html.rincondelvago.com/lenguajes-de-programacion-historia-y-evolucion.html">http://html.rincondelvago.com/lenguajes-de-programacion-historia-y-evolucion.html</a>	Historia y evolución de los lenguajes de programación
<a href="http://www.um.es/docencia/barzana/DIVULGACION/INFORMATICA/teorndprog.html">http://www.um.es/docencia/barzana/DIVULGACION/INFORMATICA/teorndprog.html</a>	Las tendencias en los lenguajes de programación
<a href="http://msdn.microsoft.com/es-es/library/x53a06bb%28v=VS.80%29.aspx">http://msdn.microsoft.com/es-es/library/x53a06bb%28v=VS.80%29.aspx</a>	Palabras claves de C#



## ANEXO 1

### Lenguajes de programación

# HISTORIA Y EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

## INTRODUCCIÓN

Los ordenadores no hablan nuestro idioma, son máquinas y, como tales, necesitan un lenguaje específico pensado por el hombre para ellas. Además, necesitan constantemente interpretar todas las instrucciones que reciben. Dada la dificultad de comunicación insalvable entre el computador y el programador, pronto aparecieron lenguajes de programación que hacen posible la comunicación con el microprocesador, utilizando términos y símbolos relacionados con el tipo de problema que se debe resolver, mediante el empleo de herramientas que brinda la informática.

Estos lenguajes permiten, por un lado, escribir las operaciones que son necesarias realizar para resolver el problema de un modo parecido a como se escribiría convencionalmente (es decir, redactar adecuadamente el algoritmo de resolución del problema) y, por el otro, se encarga de traducir el algoritmo al lenguaje máquina (proceso conocido como



compilación) con lo que se le confiere al programa la capacidad de correr (ser ejecutado) en el ordenador. El ordenador es en realidad tan sólo una máquina virtual, capaz de resolver todos los problemas que los usuarios seamos capaces de expresar mediante un algoritmo (programa).

En la actualidad hay muchos tipos de lenguajes de programación, cada uno de ellos con su propia gramática, su terminología especial y una sintaxis particular. Por ejemplo, existen algunos creados especialmente para aplicaciones científicas o matemáticas generales (BASIC, FORTRAN, PASCAL, etc. ); otros, en cambio, se orientan al campo empresarial y al manejo de textos y ficheros, es decir, son en realidad fundamentalmente gestores de información (COBOL, PL/1, etc. ), o muy relacionados con el lenguaje máquina del ordenador (como el C y el ASSEMBLER).

Los ordenadores se programaban en lenguaje máquina pero las dificultades que esto conllevaba, junto con la enorme facilidad de cometer errores, cuya localización era larga y compleja, hicieron concebir, en la década de los 40, la posibilidad de usar lenguajes simbólicos. Los primeros en aparecer fueron los ensambladores, fundamentalmente consistía en dar un nombre (mnemónico) a cada tipo de instrucción y cada dirección (etiqueta). Al principio se hacía el programa sobre papel y, después se traducía a mano con la ayuda de unas tablas, y se introducían en la máquina en forma numérica, pero pronto aparecieron programas que se ensamblaban automáticamente.

## DEFINICIONES

Es complicado definir qué es y qué no es un lenguaje de programación. Se asume generalmente que la traducción de las instrucciones a un código que comprende la computadora debe ser completamente sistemática. Normalmente es la computadora la que realiza la traducción.

A continuación, se redactan una serie de definiciones de los lenguajes de programación:

→ Un lenguaje de programación es una notación para escribir programas, a través de los cuales podemos comunicarnos con el hardware y dar así las órdenes adecuadas para la realización de un determinado proceso.

→ Un lenguaje está definido por una gramática o conjunto de reglas que se aplican a un alfabeto constituido por el conjunto de símbolos utilizados. Los distintos niveles de programación existentes nos permiten acceder al hardware, de tal forma que según utilicemos un nivel u otro, así tendremos que utilizar un determinado lenguaje ligado a sus correspondientes traductores.

→ Conjunto de normas lingüísticas (palabras y símbolos) que permiten escribir un programa y que éste sea entendido por el ordenador y pueda ser trasladado a ordenadores similares para su funcionamiento en otros sistemas.



→ Conjunto de instrucciones, órdenes y símbolos reconocibles por autómatas, a través de su unidad de programación, que le permite ejecutar la secuencia de control deseada. Al conjunto de total de estas instrucciones, órdenes y símbolos que están disponibles, se le llama lenguajes de programación del autómata.

→ El programa está formado por un conjunto de instrucciones, sentencias, bloques funcionales y grafismos que indican las operaciones a realizar. Las instrucciones representan la tarea más elemental de un programa: leer una entrada, realizar una operación, activar una salida, etc. La sentencia representa el mínimo conjunto de instrucciones o sentencias que realizan una tarea o función compleja: encontrar el valor de una función lógica en combinación de varias variables, consultar un conjunto de condiciones, etc. El bloque funcional es el conjunto de instrucciones o sentencias que realizan una tarea o función compleja: contadores, registros de desplazamientos, transferencias de información, etc. Todos estos elementos están relacionados entre sí mediante los símbolos o grafismos.

→ Es un conjunto de palabras y símbolos que permiten al usuario generar comandos e instrucciones para que la computadora los ejecute. Los lenguajes de programación deben tener instrucciones que pertenecen a las categorías ya familiares de entrada/salida, cálculo/manipulación, de textos, lógica/comparación, y almacenamiento/recuperación.



## HISTORIA

Los primeros lenguajes de programación surgieron de la idea de Charles Babagge, la cual se le ocurrió a este hombre a mediados del siglo XIX. Era un profesor matemático de la universidad de Cambridge e inventor inglés, que al principio del siglo XIX predijo muchas de las teorías en que se basan los actuales ordenadores.

Consistía en lo que él denominaba la máquina analítica, que por motivos técnicos no pudo construirse sino hasta mediados del siglo XX. Con él colaboró Ada Lovedby, quien es considerada como la primera programadora de la historia, pues realizó programas para la máquina de Babagge, empleando tarjetas perforadas. Como la máquina no llegó a construirse, los programas de Ada, lógicamente, tampoco llegaron a ejecutarse, pero si suponen un punto de partida de la programación, sobre todo si observamos que en cuanto se empezó a programar, los programadores utilizaron las técnicas diseñadas por Charles Babagge, y Ada, que consistían (entre otras), en la programación mediante tarjetas perforadas. Se dice que estos dos genios de antaño, se adelantaron un siglo a su época, lo cual describe la inteligencia de la que se hallaban dotados.

En 1823 el gobierno Británico lo apoyó para crear el proyecto de una máquina de diferencias, un dispositivo mecánico para efectuar sumas repetidas. No obstante, Babagge se dedicó al proyecto de la máquina analítica, abandonando la máquina de diferencias, basándose en la programación con tarjetas perforadas a través de la creación de Charles Jacquard (francés). Este hombre era un fabricante de tejidos y había



creado un telar que podía reproducir automáticamente patrones de tejidos, leyendo la información codificada en patrones de agujeros perforados en tarjetas de papel rígido. Entonces Babagge intentó crear una máquina que programara, con tarjetas perforadas, y efectuase cualquier cálculo con una precisión de 20 dígitos. Pero la tecnología de la época no bastaba para hacer realidad sus ideas. Si bien estas ideas no llegaron a materializarse de forma definitiva, su contribución fue decisiva, ya que los ordenadores actuales responden a un esquema análogo al de la máquina analítica. En su diseño, la máquina constaba de cinco unidades básicas:

- 1) Unidad de entrada, para introducir datos e instrucciones;
- 2) Memoria, donde se almacenaban datos y resultados intermedios;
- 3) Unidad de control, para regular la secuencia de ejecución de las operaciones;
- 4) Unidad Aritmético–Lógica, que efectúa las operaciones;
- 5) Unidad de salida, encargada de comunicar al exterior los resultados.

Charles Babbage, conocido como el "padre de la informática", no pudo completar en aquella época la construcción del computador que había soñado, pero sus ideas no fueron abandonadas, siendo la base para la construcción de los primeros computadores.



**Máquina analítica**



**Máquina diferencial**

Cuando surgió el primer ordenador, el famoso ENIAC (Electronic Numerical Integrator And Calculator), su programación se basaba en componentes físicos, es decir, que se programaba cambiando directamente el hardware de la maquina. Exactamente lo que se hacía era cambiar cables de sitio para conseguir la programación de la maquina. La entrada y salida de datos se realizaba mediante tarjetas perforadas.

## **LAS TENDENCIAS DE LOS LENGUAJES DE PROGRAMACIÓN**

Hay, al menos, dos formas fundamentales desde las que pueden verse o clasificarse los lenguajes de programación: por su nivel y por principales aplicaciones. Estas visiones están condicionadas por la visión histórica por la que ha transcurrido el lenguaje. Existen cuatro niveles distintos de lenguaje de programación:

1. Los **lenguajes declarativos** son los más parecidos al castellano o inglés. En su potencia expresiva y funcionalidad están en el nivel más alto respecto a los otros. Son fundamentalmente lenguajes de órdenes,

dominados por sentencias que expresan "lo que hay que hacer" en vez de "cómo hacerlo". Ejemplos de esto tenemos a los lenguajes estadísticos como SAS y SPSS, y los lenguajes de búsqueda en base de datos, como NATURAL e IMS. Estos lenguajes se desarrollaron con la idea de que los profesionales pudieran asimilar rápidamente el lenguaje y usarlo en su trabajo, sin necesidad de programadores o prácticas de programación.

2. Los **lenguajes de alto nivel** son los más utilizados como lenguaje de programación. Aunque no son fundamentalmente declarativos, éstos permiten que los algoritmos se expresen en un nivel y estilo de escritura, de fácil lectura y comprensión por otros programadores. Además, los lenguajes de alto nivel tienen normalmente las características de "transportabilidad". Es decir, están implementados sobre varias máquinas, de forma que un programa puede ser fácilmente "transportado" (transferido) de una máquina a otra sin una revisión sustancial. En ese sentido se llaman "independientes de la máquina". Ejemplos de estos lenguajes tenemos están PASCAL, APL y FORTRAN (para aplicaciones científicas), COBOL (para aplicaciones de procesamiento de datos), SNOBOL (para aplicaciones de procesamiento de textos), LISP y PROLOG (para aplicaciones de inteligencia artificial), C y ADA (para aplicaciones de programación de sistemas) y PL/I (para aplicaciones de propósitos generales).

3. Los **lenguajes ensambladores** y los **lenguajes máquina** son dependientes de la máquina. Cada tipo de máquina, tal como VAX de digital, tiene su propio lenguaje máquina y su lenguaje ensamblador asociado. El lenguaje ensamblador es, simplemente, una representación simbólica del lenguaje de máquina asociado, lo cual permite una



programación menos tediosa que con el anterior. Sin embargo, es necesario un conocimiento de la arquitectura mecánica subyacente para realizar una programación efectiva en cualquiera de estos niveles de lenguajes.

Los siguientes tres segmentos del programa equivalentes, exponen las distinciones básicas entre lenguajes máquina y ensambladores de alto nivel:

Como muestra este ejemplo, a más bajo nivel de lenguaje, más cerca de las características de un tipo de máquina particular, y más alejado de ser comprendido por un humano ordinario. Hay también una estrecha relación (correspondencia 1:1) entre las sentencias en lenguaje ensamblador y sus formas en lenguaje máquina codificada. La principal diferencia es que los lenguajes ensambladores utilizan símbolos (X, Y, Z, A para "sumar", M para "multiplicar"), mientras que se requieren códigos numéricos (OC1A4, etc.) para que lo comprenda la máquina.

La programación de un lenguaje de alto nivel o de un lenguaje ensamblador, requiere de algún tipo de interfaz con el lenguaje máquina para que el programa pueda ejecutarse. Las tres interfaces más comunes: un "ensamblador, un "compilador" y un "intérprete". El ensamblador y el compilador traducen el programa a otro equivalente en el lenguaje X de la máquina "residente" como un paso separado antes de la ejecución. Por otra parte, el intérprete ejecuta directamente las instrucciones en un lenguaje Y de alto nivel, sin un paso de procesamiento previo.



La compilación es, en general, un proceso más eficiente que la interpretación para la mayoría de los tipos de máquina. Esto se debe principalmente a que las sentencias dentro de un "bucle" deben ser reinterpretadas cada vez que se ejecutan por un intérprete. Con un compilador, cada sentencia es interpretada y luego traducida a lenguaje máquina solo una vez.

Algunos lenguajes son interpretados, como APL, PROLOG y LISP. El resto de los lenguajes – Pascal, FORTRAN, COBOL, PL/I, SNOBOL, C, Ada y Modula-2 – son lenguajes compilados. En algunos casos, un compilador estará utilizable alternativamente para un lenguaje interpretado (tal como LISP) e inversamente (tal como el intérprete SNOBOL4 de los laboratorios Bell). Frecuentemente la interpretación es preferible a la compilación en un entorno de programación experimental o de educación, donde cada nueva ejecución de un programa implicado un cambio en el propio texto del programa. La calidad de diagnóstico y depuración que soportan los lenguajes interpretados es generalmente mejor que la de los lenguajes compilados, puesto que los mensajes de error se refieren directamente a sentencias del texto del programa original. Además, la ventaja de la eficiencia que se adjudica tradicionalmente a los lenguajes compilados frente a los interpretados, puede pronto ser eliminado, debido a la evolución de las máquinas cuyos lenguajes son ellos mismos lenguajes de alto nivel. Como ejemplo de estos están las nuevas máquinas LISP, que han sido diseñadas recientemente por Symbolics y Xerox Corporations.

Los lenguajes de programación son tomados de diferentes perspectivas. Es importante para un programador decidir cuáles conceptos emitir o cuáles incluir en la programación. Con frecuencia, el programador es osado a usar

combinaciones de conceptos que hacen al lenguaje "DURO" de usar, de entender e implementar. Cada programador tiene en mente un estilo particular de programación, la decisión de incluir u omitir ciertos tipos de datos que pueden tener una significativa influencia en la forma en que el lenguaje es usado, la decisión de usar u omitir conceptos de programación o modelos.

Existen cinco estilos de programación:

- Orientada a Objetos.
- Imperativa: entrada, procesamiento y salidas de Datos.
- Funcional: (funciones) los datos son funciones, los resultados pueden ser un valor o una función.
- Lógica: {T, F} + operaciones lógicas (Inteligencia Artificial).
- Concurrente: aún está en proceso de investigación.

El programador, diseñador e implementador de un lenguaje de programación deben comprender la evolución histórica de los lenguajes para poder apreciar por qué presentan características diferentes. Por ejemplo, los lenguajes "más jóvenes" desaconsejan (o prohíben) el uso de las sentencias GOTO como mecanismo de control inferior, y esto es correcto en el contexto de las filosofías actuales de ingeniería del software y programación estructurada. Pero hubo un tiempo en que la GOTO, combinada con IF, era la única estructura de control disponible; el programador no dispone de algo como la construcción WHILE o un IF-THEN-ELSE para elegir. Por tanto, cuando se ve un lenguaje como FORTRAN, el cual tiene sus raíces en los comienzos de la historia de los



lenguajes de programación, uno no debe sorprenderse de ver la antigua sentencia GOTO dentro de su repertorio.

Lo más importante es que la historia nos permite ver la evolución de familias de lenguajes de programación, ver la influencia que ejercen las arquitecturas y aplicaciones de las computadoras sobre el diseño de lenguajes y evitar futuros defectos de diseño aprendiendo las lecciones del pasado. Los que estudian, se han elegido debido a su mayor influencia y amplio uso entre los programadores, así como por sus distintas características de diseño e implementación. Colectivamente cubren los aspectos más importantes con los que ha de enfrentarse el diseño de lenguajes y la mayoría de las aplicaciones con las que se enfrenta el programador.

Existen varios lenguajes que están prefijados por las letras ANS. Esto significa que el American National Standards Institute ha adoptado esa versión del lenguaje como el estándar nacional. Una vez que un lenguaje está estandarizado, las máquinas que implementan este lenguaje deben cumplir todas las especificaciones estándares, reforzando así el máximo de transportabilidad de programas de una máquina a otra.

Finalmente, la notación algebraica ordinaria influyó fuertemente en el diseño de FORTRAN y ALGOL. Por otra parte, las construcciones de largas sentencias en el idioma inglés influyeron en el desarrollo del COBOL. El cálculo lambda de Church, dio los fundamentos de la notación funcional de LISP, mientras que el algoritmo de Markov motivó el estilo de reconocimiento de formas de SNOBOL. La arquitectura de computadoras de Von Neumann, la cual fue una evolución de la máquina más antigua de

Turing, es el modelo básico de la mayoría de los diseños de computadoras de las últimas tres décadas. Estas máquinas no solo influyeron en los primeros lenguajes sino que también suministraron el esqueleto operacional sobre el que evolucionó la mayoría de la programación de sistemas.

Una discusión más directa de todos estos primeros modelos no está entre los objetivos de este texto, pero es importante mencionarlo aquí debido a su fundamental influencia en la evolución de los primeros lenguajes de programación, y por su estado en el núcleo de la teoría de la computadora. Adicional a este punto cabe mencionar lo siguiente: cualquier algoritmo que pueda describirse en inglés o castellano, puede escribirse igualmente como una máquina de Turing (máquina de Von Neumann), un algoritmo de Markov o una función recursiva. Esta sección, conocida ampliamente como "Tesis de Church", nos permite escribir algoritmos en distintos estilos de programación (lenguajes) sin sacrificar ninguna medida de generalidad, o potencia de programación en la transición.

## **EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN**

Atendiendo al desarrollo de los lenguajes desde la aparición de las computadoras, que sigue un cierto paralelismo con las generaciones establecidas en la evolución de las mismas, tenemos lo siguiente:

- ✓ 1ª generación. Lenguajes máquina y ensambladores.
- ✓ 2ª generación. Primeros lenguajes de alto nivel imperativo (FROTRAN, COBOL).
- ✓ 3ª generación. Lenguajes de alto nivel imperativo. Son los más utilizados y siguen vigentes en la actualidad (ALGOL 8, PL/I, PASCAL, MODULA)
- ✓ 4ª generación. Orientados básicamente a las aplicaciones de gestión y al manejo de bases de datos (NATURAL, SQL).
- ✓ 5ª generación. Orientados a la inteligencia artificial y al procesamiento de los lenguajes naturales (LISP, PROLOG).

Para una mejor comprensión del tema, se definirán los siguientes términos:

**Programa:** conjunto de instrucciones escritas en un lenguaje de programación que indican a la computadora la secuencia de pasos, para resolver un problema.

**Código fuente:** está creado en algún lenguaje de alto nivel, por lo que es entendido 100% por el ser humano. Este debe estar complementado por su documentación o manuales donde se indica el desarrollo lógico del mismo.

**Código objeto:** es creado por los compiladores y nos sirve como enlace entre el programa fuente y el ejecutable.

### **Evolución de los lenguajes de programación**



PERIODO	INFLUENCIAS	LENGUAJES
1950 – 55	Ordenadores primitivos	Lenguajes ensamblador Lenguajes experimentales de alto nivel
1956 – 60	Ordenadores pequeños caros y lentos Cintas magnéticas Compiladores e interpretes Optimización del código	FORTTRAN ALGOL 58 y 60 COBOL LISP
1961 – 65	Ord. grandes y caros Discos Magnéticos Sistemas operativos Leng. de propósito general	FORTTRAN IV COBOL 61 Extendido ALGOL 60 Revisado SNOBOL APL ( como notación sólo)
1966 – 70	Ordenadores de diferentes tamaños, velocidades, costes Sistemas de almacenamiento masivo de datos (caros) S.O. multitarea e interactivo Compil. con optimización Leng. estandard flexibles y generales	PL/I FORTTRAN 66 (estándar) COBOL 65 (estándar) ALGOL 68 SNOBOL4 SIMULA 67 BASIC APL/360
1971 – 75	Micro ordenadores Sistemas de almacenamiento masivo de datos pequeños y baratos	PASCAL COBOL 74 PL /I



	Progr. Estructurada Ingeniería del software Leng. sencillos	
1976 – 80	Comp. baratas y potentes Sistemas distribuidos Prog. tiempo–real Prog. interactiva Abstracción de datos Prog. con fiabilidad y fácil mantenimiento	ADA FORTRAN 77 PROLOG C

Todo este desarrollo de las computadoras y de los lenguajes de programación, suele divisarse por generaciones y el criterio que se determinó para indicar el cambio de generación no está muy bien definido (debido a que no hubo un cambio significativo en diseño estructural del software sino más bien fue paulatino), pero resulta aparente que deben cumplirse al menos los siguientes requisitos:

- La forma en que están construidas.
- Forma en que el ser humano se comunica con ellas.



## ANEXO 2

### El paradigma orientado a objetos

#### INTRODUCCIÓN

##### 1) Programación orientada a objetos (P.O.O.)

Un proyecto de software es complejo. Las GUI<sup>1</sup>, acceso transparente a datos y capacidad de trabajo. En red, lo hacen más complejo aún. Para enfrentarse a esta complejidad, nace la POO.

##### 2) ¿Qué es la POO?

Es una técnica o estilo de programación que utiliza objetos como bloque fundamental de construcción.

##### 3) Elementos básicos de la POO

###### ♦ Bloques

Son un conjunto complejo de datos (atributos) y funciones (métodos) que poseen una determinada estructura y forman parte de una organización. Los atributos definen el estado del objeto; los métodos, su comportamiento.

---

<sup>11</sup> Interfaz gráfica del usuario (graphical user interface)



#### ♦ **Métodos**

Es un programa procedimental que está asociado a un objeto determinado y cuya ejecución solo puede desencadenarse a través del mensaje correspondiente.

#### ♦ **Mensajes**

Es una petición de un objeto a otro para que éste se comporte de una manera determinada, ejecutando uno de sus métodos. Los mensajes comunican a los objetos con otros y con el mundo exterior. A esta técnica de enviar mensajes se le conoce como *paso de mensajes*.

#### ♦ **Clases**

Es un modelo o contenedor definido por el usuario que determina la estructura de datos y las operaciones asociadas con ese tipo.

### **4) Características**

#### ♦ **Abstracción**

Significa extraer las propiedades esenciales de un objeto que lo distinguen de los demás tipos de objetos y proporciona fronteras conceptuales definidas, respecto al punto de vista del observador. Es la capacidad para encapsular y aislar la información de diseño y ejecución.

#### ♦ **Encapsulamiento**

Es el proceso de almacenar en un mismo compartimiento (una caja negra) los elementos de una abstracción (toda la información relacionada con un objeto) que constituyen su estructura y su comportamiento. Esta

información permanece oculta tanto para los usuarios como para otros objetos y puede ser accedida solo mediante la ejecución de los métodos adecuados.

#### ♦ Herencia

Es la propiedad que permite a los objetos construirse a partir de otros objetos. La clase base contiene todas las características comunes. Las sub-clases contienen las características de la clase base más las características particulares de la sub-clase. Si la sub-clase hereda características de una clase base, se trata de herencia simple. Si hereda de dos o más clases base, herencia múltiple.

#### ♦ Polimorfismo

Literalmente significa "cualidad de tener más de una forma". En POO, se refiere al hecho que una misma operación puede tener diferente comportamiento en diferentes objetos. En otras palabras, diferentes objetos reaccionan al mismo mensaje de modo diferente.

### 5) Ventajas

#### ♦ Modelos

La POO permite realizar un modelo de sistema casi independientemente de los requisitos del proyecto. La razón es que en la POO la jerarquía la establecen los datos, en cambio en la programación estructurada, la jerarquía viene definida por los programas. Este cambio hace que los modelos se establezcan de forma similar al razonamiento humano y, por lo tanto, resulte más natural.



#### ♦ **Modularidad**

Un programa es modular si se compone de módulos independientes y robustos. Esto permite la reutilización y facilita la verificación y depuración de los mismos. En POO, los módulos están directamente relacionados con los objetos. Los objetos son módulos naturales ya que corresponden a una imagen lógica de la realidad.

#### ♦ **Extensibilidad**

Durante el desarrollo de sistemas, ocurre la aparición de nuevos requisitos, por eso es deseable que las herramientas de desarrollo permitan añadirlos sin modificar la estructura básica del diseño. En POO es posible lograr esto siempre y cuando se hayan definido de forma adecuada la jerarquía de clases, los atributos y métodos.

#### ♦ **Eliminación de redundancia**

En el desarrollo de sistemas se desea evitar la definición múltiple de datos y funciones comunes. En POO esto se logra mediante la herencia (evita la definición múltiple de propiedades comunes a muchos objetos) y el polimorfismo (permite la modificación de métodos heredados). Solo hay que definir los atributos y los métodos en el antepasado más lejano que los comparte.

#### ♦ **Reutilización**

La POO proporciona un marco perfecto para la reutilización de las clases. El encapsulamiento y la modularidad nos permiten utilizar una y otra vez las mismas clases en aplicaciones distintas. En efecto, el aislamiento entre



distintas clases significa que es posible añadir una nueva clase o un módulo nuevo (extensibilidad) sin afectar al resto de la aplicación.

## 6) Lenguajes en POO

### ♦ Puros

Son los que solo permiten realizar programación orientada a objetos. Ejemplo: smalltalk, java.

### ♦ Híbridos

Son los que permiten la POO con la programación estructurada. Ejemplo: C++, pascal.

## 7) POO en C++ y java

### a) Tipos de clases

Una de las principales decisiones al trabajar con POO es la de selección de clases. Existen 4 tipos:

**1. Manejadoras de datos o de estados:** su responsabilidad principal es mantener información de datos o estado. Se reconocen como los sustantivos en la descripción de un problema y generalmente son los bloques de construcción más importantes de un diseño.

**2. Pozos o fuentes de datos:** estas clases generan datos o los aceptan para procesarlos mas adelante. A diferencia de los anteriores, estas clases



no retienen los datos por un periodo de tiempo, sino que los genera sobre demanda o los procesa cuando se le llama.

**3. Vistas:** se encargan de la presentación de la información.

**4. Auxiliares o de ayuda:** guardan poca o ninguna información de estado, pero que asisten en la ejecución de tareas complejas.

### b) Sintaxis de una clase

C++:

```
Class nombre_clase [: [public/protected/private]
clase_madre]
{
    [lista de atributos];
    [lista de metodos];
};
```

Java:

```
[public] [final/abstract] class nombre_clase [extends
clase_madre]
[implements interfaz1,
[interface2,...]...]
{
    [lista de atributos];
    [lista de metodos];
};
```

### c) Modificadores de acceso a miembros de clases



Existen 3 tipos de usuarios de una clase:

- La propia clase.
- Usuarios genéricos (otras clases, métodos, etc.)
- Clases derivadas.

Cada usuario tiene distintos privilegios o niveles de acceso.

C++:

▪ *Private*

Por defecto todo lo declarado dentro de la clase es privado y solo puede ser accedido por Funciones miembro o por funciones amigas.

▪ *Public*

Pueden ser accedidos por funciones miembro y no miembro de una clase.

▪ *Protected*

Pueden ser accedidos por función miembro, por funciones amigas o por funciones miembro de sus Clases derivadas.

Java:

▪ *Private*

Solo puede ser accedida por métodos propios de la clase.

▪ *Private-protected*

Pueden ser accedidos por las sub-clases, sin importar el paquete al que pertenezcan. Sin embargo, las sub-clases solo pueden modificar estos atributos para objetos de la sub-clase, no de la clase Madre.

▪ *Protected*

Permite el acceso a las sub-clases y a las clases del mismo paquete.



- *Friendly*

Es el valor por defecto. Permite el acceso solo a clases del mismo paquete.

- *Public*

Pueden ser accedidas por todos.

#### **d) Atributos**

Las variables declaradas dentro de un método son locales a él; las declaradas en el cuerpo de la clase son miembros de ella y son accesibles por todos los métodos de la clase. Los atributos miembros de la clase pueden ser: atributos de clase (declarado como static) o atributos de instancia.

C++:

[static]tipo\_dato nombre\_dato; (no se puede inicializar un dato miembro de una clase)

Java:

[modificador\_de\_acceso] [static] [final] [transient] [volatile] tipo\_dato nombre\_dato [= valor];

- *Final*

Implica que un atributo no puede ser sobre-escrito o redefinido, es decir que no se trata de una Variable sino de una constante.

- *Transient*

Son atributos que no se graban cuando se archiva un objeto, o sea no forman parte permanente del estado de éste.

▪ *Volatile*

Se utiliza cuando la variable puede ser modificada por distintos *threads*. Básicamente implica que varios *threads* pueden modificar la variable en forma simultánea, y *volatile* asegura que se vuelva a leer la variable, por si fue modificada, cada vez que se la vaya a usar.

## e) Métodos

Dentro de los métodos pueden incluirse:

- ✓ Declaraciones de variables locales.
- ✓ Asignaciones a variables.
- ✓ Operaciones matemáticas.
- ✓ Llamados a otros métodos.
- ✓ Estructuras de control.
- ✓ Excepciones.

C++:

Se puede declarar y definir dentro de la clase:

```
Class nombre_clase
{
    tipo_dato dato;
```



```
[modificador_de_acceso]:
    tipodato_devuelto
nombre_metodo(parametros)
    {
        cuerpo_metodo;
    };
}
```

También, se puede declarar dentro de la clase y definirlo fuera:

```
Class nombre_clase
{
    tipo_dato dato;
    [modificador_de_acceso]:
        tipodato_devuelto
nombre_metodo(parametros);
};
```

```
Tipodato_devuelto nombre_clase :: nombre_metodo(parametros)
{
    cuerpo_metodo;
}
```

Java:

En java se debe declarar y definir la función dentro de la clase:

```
Class nombre_clase
{
```



```
    tipo_dato dato;  
    [modificador_de_acceso]    [static]    [abstract]  
[final] [native] [synchronized] tipodato_devuelto  
        nombre_metodo  
(parametros)[throws excepcion1 [,excepcion2]]  
        {  
            cuerpo_metodo;  
        }  
}
```

- **Static**

Al igual que con los atributos, un método declarado como static es compartido por todas las Instancias de la clase.

- **Abstract**

Son aquellos de los que se da la declaración pero no la implementación, para generar una clase Abstracta.

- **Final**

Es un método que no puede ser redefinido por las sub-clases herederas.

- **Native**

Es un método que esta escrito en otro lenguaje que no es java.

- **Synchronized**

Permite sincronizar varios threads para el caso en que dos o más quieran acceder en forma Concurrente.



- Throws

Sirve para indicar que la clase genera determinadas excepciones.

- Llamada a métodos

Se llama a un método de la misma clase simplemente con el nombre del método y los parámetros entre paréntesis.

Cuando se necesita llamar a un método de un objeto de otra clase, se utiliza:

Nombre\_objeto.nombre\_metodo(parametros)

## **f) Métodos o funciones miembros**

Funciones simples (c++ y java)

Son los vistos anteriormente, se pueden definir dentro o fuera de la clase.

Funciones en línea (inline) (c++)

Se crean definiendo la función dentro de la clase (declaración implícita) o definiendo la función fuera de la clase pero anteponiendo la palabra reservada inline (declaración explícita).

Funciones constructores (c++ y java)

Es una función especial que sirve para inicializar objetos de una clase. En general, tienen el mismo nombre de la clase que inicializa, no devuelven valores, pueden admitir parámetros, pueden existir más de un constructor (e incluso no existir), si no se define ningún constructor el compilador genera uno por defecto, se llaman al momento de crear un objeto.



### Constructores por defecto

Es un constructor que no acepta argumentos. Si no se define, el compilador genera uno que asigna espacio de memoria y lo pone en cero.

### Constructores con argumentos

Inicializan un objeto con los valores del argumento. Pueden existir varios que se diferencian por la cantidad y tipo de argumentos.

### Constructores copiadores

Crea un objeto a partir de uno existente. Toma como único parámetro otro objeto del mismo tipo. Si no se declara el compilador genera uno por defecto asignándole al nuevo objeto los valores del objeto a la izquierda del operador =.

En java se puede inicializar el objeto tras la creación física del mismo, asignándole los valores que se le dará.

### Función destructor (c++)

Cumplen la función inversa a la del constructor, eliminando el espacio de almacenamiento que ocupó el objeto al ser creado. Características: tienen el mismo nombre de la clase, van precedidos por el carácter ~, solo existe un destructor por clase, no admiten argumentos, no retornan ningún valor, el compilador llama a un destructor cuando el objeto sale fuera de ámbito.

### Funciones amigas (c++)

Es una función no miembro que puede acceder a las partes private de una clase. Se declaran anteponiendo la palabra reservada friend. También se pueden declarar como amigas a las clases. En este caso todas las

funciones de la clase amiga pueden acceder a las partes privadas de la otra clase. Funciones sobrecargadas (c++ y java) es una función que tiene más de una definición. Las funciones sobrecargadas tienen el mismo nombre, pero deben tener un número distinto de argumentos o diferentes tipos de argumentos. Las definiciones operan sobre funciones distintas.

Las únicas funciones miembro que no se pueden sobrecargar son los destructores.

Funciones operador (c++)

Permiten sobrecargar un operador existente (ej: suma) para utilizarlos con objetos. Se declara poniendo la palabra reservada `operator` seguida por el operador específico y la lista de argumentos y el cuerpo de la función.

Restricciones:

- El operador debe ser uno válido para c++ y no se puede cambiar el símbolo.
- Funciona solo al utilizar objetos de clase.
- No se puede cambiar la asociación de los operadores, es decir para que sirven.
- No se puede cambiar un operador binario para funcionar con un único objeto.
- No se puede cambiar un operador unario para que funcione con dos objetos.

g) Objetos



### **C++:**

Se pueden crear en forma estática (nombre\_clase objeto1) o dinámica (nombre\_clase \*objeto2; Objeto2= new nombre\_clase;)

### **Java:**

Solo se crean los objetos en forma dinámica: nombre\_clase objeto1=new nombre\_clase();

### **h) Arrays de objetos**

Se pueden crear arrays de objetos de la misma forma que se crea un array normal.

#### **C++:**

Estatico: nombre\_clase objeto[10];

Dinamico: nombre\_clase \*objeto;

Objeto=new nombre\_clase[10];

#### **Java:**

Nombre\_clase objeto=new nombre\_clase[10];

### **i) Puntero this**

Es un puntero al objeto asociado con la invocación de una función miembro. Normalmente no se explicita ya que el lenguaje realiza esta operación transparente y en forma automática. Igualmente hay casos en los que se debe usar el puntero this explícitamente:



- Como argumento en una llamada a una función para pasar un puntero al objeto asociado con La invocación de la función miembro.

### **Ej: f(this);**

Hacer una copia del objeto asociado con la invocación o asignar un nuevo valor al objeto.

Ej: void t::g (t &a, t &b)

```
{  
    ...  
    a= *this;  
    ...  
    *this= b;  
}
```

Devolver una referencia al objeto asociado con la invocación de la función miembro o

### **Constructor**

Ej: t& t::f (int a)

```
{  
    ...  
    return *this;  
}
```

### **C++:**

This->nombre\_objetomiembro



**Java:**

This.nombre\_objetomiembro

## **2. HERENCIA Y POLIMORFISMO**

### **1) Herencia**

Herencia es la propiedad de que los ejemplares de una clase hija extiendan el comportamiento y los datos asociados a las clases paternas. La herencia es siempre transitiva, es decir que una sub-clase hereda características de superclases alejadas muchos niveles.

### **2) Beneficios de la herencia**

Reusabilidad del software cuando el comportamiento se hereda de otra clase, no es necesario reescribir el código que lo define.

#### **a) Compartición de código**

Muchos usuarios o proyectos distintos pueden usar las mismas clases. Por otro lado, la herencia reduce el tiempo de escritura y el tamaño final del programa.

#### **b) Consistencia de la interfaz**

El comportamiento de una clase madre que heredan todas sus clases hijas será el mismo, de esta manera se asegura que las interfaces para objetos

serán similares y no solo un conjunto de objetos que son parecidos pero que actúan e interactúan de forma diferente.

### **c) Componentes de software**

La herencia nos permite escribir componentes de software reutilizables.

### **d) Modelado rápido de prototipos**

Cuando un sistema se construye casi totalmente de componentes reutilizables, se puede dedicar más tiempo a la realización de aquellas partes nuevas. A este estilo de programación se lo conoce como 'modelado rápido de prototipos o programación exploratoria'.

### **e) Ocultación de información**

Un programador puede reutilizar un componente conociendo solamente la naturaleza e interfaz del mismo y sin la necesidad de conocer los detalles técnicos empleados para su realización.

### **f) Polimorfismo**

Permite al programador generar componentes reutilizables de alto nivel que puedan adaptarse a diferentes aplicaciones mediante el cambio de sus partes de bajo nivel.

## **3) Heurísticas para crear sub-clases**

Para saber si una clase debe convertirse en una subclase de otra mediante la herencia, hay que aplicar la regla del es-un y es-parte-de.



### **a) Es-un**

Se dan entre dos conceptos cuando el primero (la sub-clase) es un ejemplar especificado del segundo (la clase base). Ej: un avión es un transporte.

### **b) Es-parte-de**

Se da entre dos conceptos cuando el primero (la sub-clase) es una parte del segundo (la clase base), sin ser ninguno, en esencia, la misma cosa. Ej: un motor es parte de un auto.

### **c) Distintos tipos de herencia**

#### **→ Especialización**

Es la forma de herencia mas común y cumple en forma directa la regla es-un.

#### **→ Especificación**

Se trata de un caso especial de sub-clasificación por especialización, excepto que las sub-clases no son refinamientos de un tipo existente, sino más bien realizaciones de una especificación incompleta. Es decir, que la superclase describe un comportamiento que será implantado solo por las sub-clases.

#### **→ Construcción**

Se da cuando la sub-clase ha heredado casi completamente su comportamiento de la superclase y solo tiene que modificar algunos métodos o los argumentos de cierta manera.

→ **Generalización**

Esta es la opuesta a la creación de sub-clases por especialización. Se debe evitar; solo deben aplicarse cuando no se pueden modificar las clases existentes o se deben anular métodos de las mismas.

→ **Extensión**

Agrega una capacidad totalmente nueva a un objeto existente. Se distingue de la generalización, ya que esta debe anular al menos un método de la clase base, mientras que la extensión solo agrega métodos nuevos.

→ **Limitación**

Es una variante de la especificación en donde el comportamiento de la sub-clase es más reducido o está más restringido que el comportamiento de la superclase. También se da en situaciones en las que las clases existentes no pueden modificarse.

→ **Variación**

Se da cuando dos o más clases tienen implantaciones similares, pero no parece haber ninguna relación jerárquica entre los conceptos representados por las clases. Ej: código del mouse y de la placa de video.

→ **Combinación**

Se da cuando una sub-clase resulta de la combinación de características de dos o más clases.

#### **4) Herencia y jerarquía de clases**

C++ y java utilizan un sistema de herencia jerárquica. Una clase hereda de otra, creando así nuevas clases a partir de clases existentes. Solo se pueden heredar clases, no funciones ordinarias ni variables. Una sub-clase deriva de una clase base. La clase derivada puede a su vez ser utilizada como clase base para derivar más clases, conformándose así la jerarquía de clases.

#### **Características de las clases derivadas**

Una clase derivada o sub-clase:

- ♦ Puede a su vez ser una clase base, dando lugar a la jerarquía de clase.
- ♦ Los miembros heredados por una clase derivada, pueden a su vez ser heredados por más clases derivadas a ella.
- ♦ Hereda todos los miembros de la clase base, pero solo podrá acceder a aquellos que los especificadores de acceso de la clase base lo permitan. Las clases derivadas solo pueden acceder a los miembros public, protected y private-protected (en java) de la clase base, como si fueran miembros propios.
- ♦ No tienen acceso a los miembros private de la clase base.
- ♦ Pueden añadir sus propios datos y funciones miembros.

Los siguientes elementos de la clase base no se heredan:

- ♦ Constructores y destructores.



- ♦ Funciones y datos estáticos de la clase.
- ♦ Funciones amigas y operadores sobrecargados (solo c++).

Sintaxis de una clase derivada

### **C++:**

```
Class clase_derivada : [public / protected / private] clase_base [, [public /  
protected / private]clase_base2]
```

```
{  
    cuerpo clase derivada;  
};
```

### **Java:**

```
Class nombre_derivada extends nombre_base
```

```
{  
    cuerpo clase derivada;  
}
```

### **Especificadores de acceso**

C++:

Si no se especifica el tipo de derivación, c++ supone que el tipo de herencia es private.

Si en la derivación especificamos:

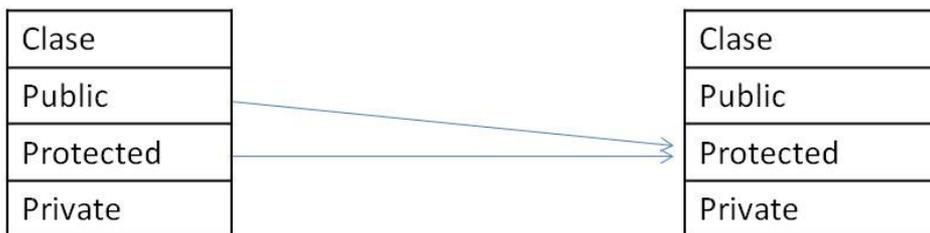
#### **♦ Public**

Los miembros public pasan a ser public en la clase derivada. Los miembros protected pasan a ser protected. Y los privados permanecen privados en la clase base.



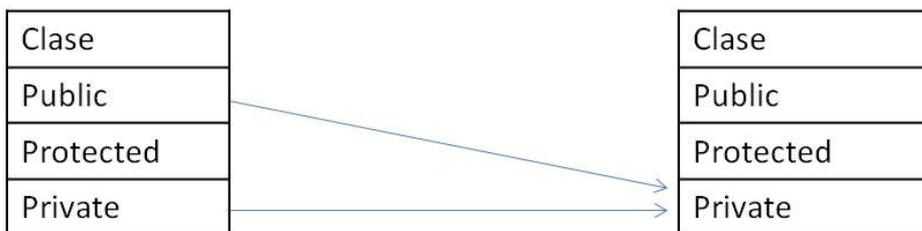
♦ **Protected**

Todos los miembros public y protected de la clase base son miembros protected en la derivada.



♦ **Private**

Todos los miembros public y private son miembros private en la clase derivada.





**Java:**

Para analizar los tipos de herencia en java hay que tener en cuenta que existen 5 distintos tipos de clases:

- Clase base.
- Clases derivadas del mismo paquete.
- Clases derivadas en distinto paquete.
- Clases del mismo paquete.
- Clases en distinto paquete.

	Miembros declarados como...				
	Private	Private-protected	Protected	Friendly	Public
Clase base.	S	S	S	S	S
Clases derivadas del mismo paquete.	N	R	S	S	S
Clases del mismo paquete.	N	N	S	S	S
Clases derivadas en distinto paquete.	N	R	R	N	S
Clases en distinto paquete.	N	N	N	N	S

Tipo de acceso:

S: acceso posible.

N: acceso denegado.

R: reservado a las subclases. Acceso posible si el miembro

se refiere a un objeto de la subclase y no a uno de la clase base.

### *Redefinición de métodos*

Cuando se hace heredar una clase de otra, se pueden redefinir ciertos métodos a fin de refinarlos, o bien de modificarlos. El método lleva el mismo nombre y la misma signatura, pero solo se aplica a objetos de la sub-clase o sus descendientes.

### *Constructores*

Cuando se construye una clase hija, es necesario también llamar al constructor de la clase madre. Esta invocación puede ser implícita o explícita.

Es implícita si el constructor de la madre no toma parámetros. En este caso, ya sea que el constructor de la hija sea implícito o no, la llamada al constructor de la madre es automática.

Si el constructor de la madre necesita parámetros, hay que pasárselos. Es necesario definir un constructor en la hija y este debe llamar al constructor de la madre enviando los parámetros.

Para realizarlo se sigue el siguiente orden:

- Llamar al constructor de la clase madre.
- Construir los miembros de la clase hija.



- Ejecutar las instrucciones contenidas en el cuerpo del constructor de la clase hija.

### **C++:**

Derivada :: derivada (tipo1 x, tipo2 y) : base (x,y) [, base2 (x,y) ]

```
{  
    cuerpo del constructor;  
}
```

En herencia múltiple, el orden real de invocación de constructores se da de acuerdo al orden en que fueron declaradas las clases. C++ utiliza el siguiente orden de inicialización:

- Primero, se inicializan todas las clases bases virtuales.
- Las clases bases no virtuales se inicializan en el orden en que aparecen en la declaración de clases.
- Por último, se ejecuta el constructor de la clase derivada.

### **Java:**

En la definición del constructor de la clase derivada se utiliza la palabra super para simbolizar la llamada al constructor de la clase base.

```
Class hija extends madre  
{  
    public hija (tipo1 x)  
    {  
        super (y);  
    }  
}
```

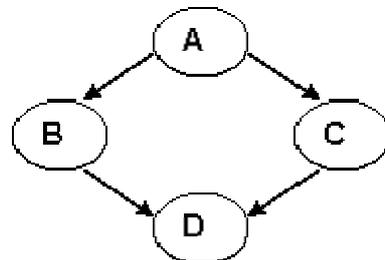
```
        cuerpo del constructor;  
    ]  
}
```

### *Destructores*

Al contrario que con los constructores, una función destructor de una clase derivada se ejecuta antes que el destructor de la clase base (los destructores no se heredan).

### *Clases bases virtuales*

Es una clase que es compartida por otras clases base con independencia del número de veces que esta se produce en la jerarquía de derivación.



Supongamos que la clase base A, tiene sus propios miembros. Definimos dos clases derivadas de ésta, B y C, también con sus propios métodos y los heredados. Definimos también una clase D, derivada de B y C (derivación múltiple). Este tipo de jerarquía puede dar lugar a problemas ya que la clase D heredará las funciones miembro de B y C, pero también derivará las de A dos veces.

Para evitar este problema, debemos definir a las clases base como clases virtuales, lo cual significa que solo una copia de los miembros ambiguos pasara a la clase derivada.

Class a

{...}

Class b : virtual public a

{...}

Class c : virtual public a

{...}

Class d : public b , public c

{...}

## 5) Polimorfismo

Consiste en llamar a un método según el tipo estático de una variable, basándose en el núcleo para llamar a la versión correcta del método. El polimorfismo se realiza por estos métodos:

→ Sobrecarga de funciones (c++ y java)

En este caso el compilador determina qué función debe utilizarse en tiempo de compilación ya que conocerá el objeto implicado. Este tipo de ligadura se la conoce como ligadura temprana, estática o previa.

Cuando existen punteros implicados, el compilador no sabe a que objeto se está referenciando. En este caso se produce una ligadura dinámica.

→ Con funciones virtuales (c++)

Cuando varias sub-clases derivan de una clase base, éstas pueden emplear las funciones que heredan de la misma forma, pero otras pueden requerir

elementos adicionales o incluso formatos totalmente nuevos. Para enfrentar esto, existen 3 soluciones posibles:

- a) Definir las funciones con distintos nombres (es fácilmente realizable pero no es ideal porque genera complejidad).
- b) Sobrecargar la función. Esto es útil cuando la jerarquía de clases es pequeña.
- c) Identificar la función miembro de la clase base, que se puedan llegar a utilizar en clases derivadas y declararlas como virtuales.

En este último caso, la ligadura se lleva a cabo en tiempo de ejecución.

La función debe ser declarada como virtual en la primera clase en la que está presente. La palabra clave virtual permite definir la función bajo el mismo nombre tanto en la clase base como en la derivada. Al utilizar funciones virtuales se pueden utilizar punteros a la clase base para referenciar objetos de una clase derivada.

### **Función virtual pura**

Es una función virtual declarada en una clase base que no está definida y no tiene cuerpo. Deben definirse luego en la clase derivada.

#### **C++:**

Virtual tipo nombre\_funcion (parametros) = 0;

#### **Java:**

Se llaman métodos abstractos y se declaran pero no se escribe su implementación.



Abstract metodoabstracto ();

### **Clases abstractas**

Son clases que se diseñan para ser heredadas y solo se pueden utilizar como clases base. No pueden ser instanciadas.

#### **C++:**

Una clase es abstracta si tiene al menos una función virtual pura.

#### **Java:**

Deben declararse como abstract.

```
Abstract nombre_clase  
{  
    abstract void metodo_abstracto ();  
}
```

Pasos para obtener polimorfismo en C++

1. Crear una jerarquía de clases con las funciones miembro importantes definidas como virtuales. Si las clases base son tales que no se pueden implementar estas funciones en ellas, declarar funciones virtuales puras.
2. Proporcionar implementaciones concretas de clases virtuales en las clases derivadas. Cada clase derivada tiene su propia versión de las funciones.
3. Manipular instancias de las clases derivadas a través de punteros.



SUAYED  
UNA OPCIÓN  
PARA TI

Licenciatura: Informática

# UNIDAD 2 TIPOS DE DATOS ELEMENTALES

(VARIABLES, CONSTANTES, DECLARACIONES,  
EXPRESIONES Y ESTRUCTURA DE UN  
PROGRAMA)

## APUNTES DIGITALES PLAN 2012



SUAYED  
UNA OPCIÓN  
PARA TI



## OBJETIVO ESPECÍFICO

Al terminar la unidad, el alumno deberá conocer los componentes básicos de la programación y la estructura de un programa.



SUAYED

## 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.



## LO QUE SÉ

Define con tus propias palabras los siguientes términos:

1. Tipo de dato.
2. Constante.
3. Variable.



SUAYED  
Sistema Universitario  
Autónomo de Uruguay

## TEMARIO DETALLADO

**(6 horas)**

- 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



## 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.

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	2	Para números



	32.767		enteros.
unsigned int	0 a 65.535	2	Para números enteros.
long int	$\pm 2.147.483.647$	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.

```
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.

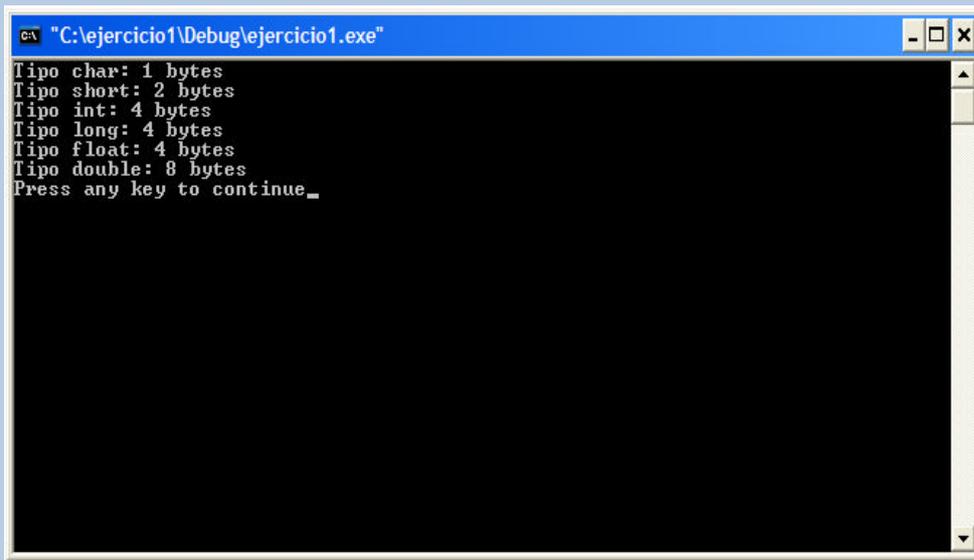
### Ejercicio 1.1

```
# 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.

- ♦ `if` Determina si se ejecuta o no una sentencia o grupo de sentencias.
- ♦ `for` Ejecuta un número de sentencias un número determinado de veces.
- ♦ `while` Ejecuta un número de sentencias un número indeterminado de veces
- ♦ `return` Devuelve el valor de una función.
- ♦ `int` Determina que una variable sea de tipo entero.
- ♦ `void` Indica que una función no devuelve valor alguno.

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 3).

## 2.5. Expresiones y reglas de prioridad

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

*Ejemplo:*

`s = s + i`

`n == 0`

`++i`

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, y viceversa, si la expresión es falsa el resultado es verdadero.

→ Prioridades de los operadores aritméticos

Son prioridades de los operadores matemáticos:

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.

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

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.  $2.1 * (1.5 + 3.0 * 4.1) = 2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98$   
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:

Ejemplos:

numéricas: -7 3.1416 -2.5e-3

caracteres: 'a' '\n' '\0'

cadenas: "indice general"

SINTAXIS:

```
const tipo nombre=valor_entero;
```

```
const tipo nombre=valor_entero.valor_decimal;
```

```
const tipo nombre='carácter';
```

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 6.1.

```
#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:



```
C:\Archivos de programa\Microsoft Visual Studio\MyProjects\ejemplo1\Debug\ejemplo1.exe
El valor del Dolar es 11.50 pesos
El valor del Peso es 1
Esto es una prueba
Ejemplo de constantes y defines_
```

## 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.

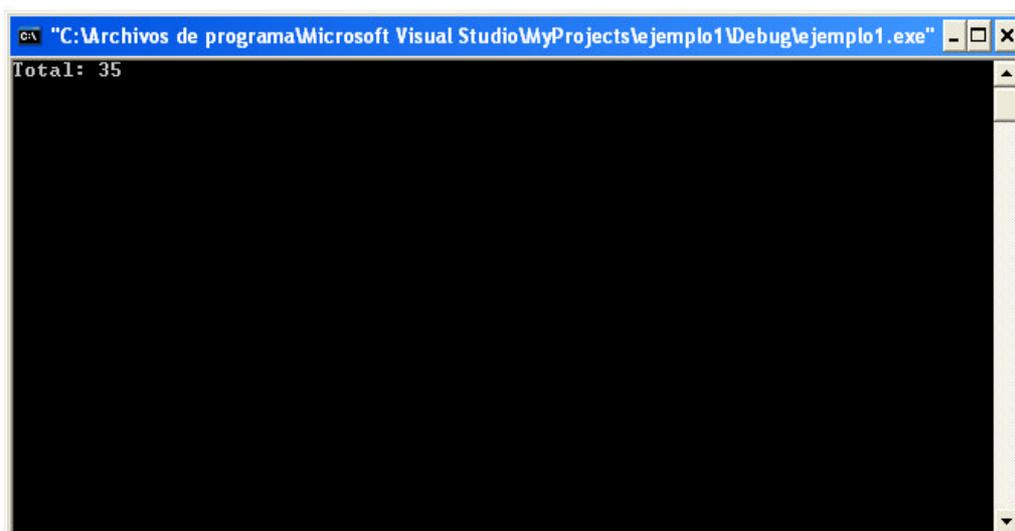


### Ejercicio

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
    float num1=25.75, num2=10.15;
    int total=0;

    total=(int)num1+(int)num2;
    printf("Total: %d",total);
    getch();
}
```

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. Las variables pueden ser:



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;
}
```

Las variables pueden ser:

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.

```
float num1=25.75, num2=10.15; <-DECLARACION DE VARIABLES
int total=0; <-DECLARACION DE VARIABLES
```

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



```
total=(int)num1+(int)num2; <-DESARROLLO DEL PROGRAMA  
printf("Total: %d",total); <-DESARROLLO DEL PROGRAMA  
getch(); <-DESARROLLO DEL PROGRAMA
```

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

```
} LLAVES DE FIN
```

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



## RESUMEN DE LA UNIDAD

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: son globales y se pueden usar desde cualquier parte del programa si se declaran antes de la función main, son locales si se declaran dentro de una función y sólo se pueden usar dentro de la misma función. También hay variables de registro (almacenadas en los registros internos del microprocesador), estáticas (variables locales que mantienen su valor entre las llamadas a funciones) y externas (variable global que es conocida por todos los módulos que conforman el programa).

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



## GLOSARIO DE LA UNIDAD

### **Constante**

Un valor que no cambia durante la ejecución del programa.

### **Dato**

Elemento más pequeño con un significado propio.

### **Expresión**

Conjunto de variables y constantes (operandos) y operadores.

### **Identificador**

Serie de caracteres que identifican a una variable o constante de un programa.

### **Palabra reservada**

Son un conjunto de caracteres que tienen un significado especial dentro del lenguaje de programación, y no pueden ser usados para otro fin.

### **Tipo de dato**

Un tipo de dato determina el rango de valores que puede poseer una variable, algunos ejemplos son int (entero), char (carácter), bool (booleano), string (cadena), double (doble), etcétera.



## **Variable**

Un valor que cambia durante la ejecución del programa.

## **Variables 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. Cabe hacer notar que declarándolas dentro de la función main, las variables son locales pero como es la función principal su ámbito también abarca a todo el programa.

## **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.



## ACTIVIDADES DE APRENDIZAJE

### ACTIVIDAD 1

Investiga los tipos de datos que se emplean en el lenguaje C++ y compáralos con los empleados con los usados en C. Desarrolla un cuadro comparativo con los tipos de datos en C y C++.

### ACTIVIDAD 2

Investiga los tipos de datos que se emplean en el lenguaje Java. Realiza un cuadro comparativo de similitudes y diferencias de 5 tipos de datos entre los lenguajes C++ y Java, agrega la sintaxis de la forma de declararlos y utilizarlos.

### ACTIVIDAD 3

Investiga las siguientes funciones:

- Función de la palabra reservada enum.
- Función de los identificadores.



**ACTIVIDAD 4**

Investiga y enuncia las palabras reservadas del lenguaje PHP.

**ACTIVIDAD 5**

Elabora un programa en C que determine si un número es par o impar.

**ACTIVIDAD 6**

Elabora un programa en C que determine si dos números introducidos por el usuario son iguales.

**ACTIVIDAD 7**

Investiga e indica cuál es la prioridad del operador de exponente en el lenguaje Visual Basic.

**ACTIVIDAD 8**

Investiga y describe el uso de variables locales y globales en Python.

**ACTIVIDAD 9**

Investiga y describe el uso de constantes en Java.



### ACTIVIDAD 10

Investiga y escribe la estructura de un programa escrito en Visual Basic.



# CUESTIONARIO DE REFORZAMIENTO

*Contesta el siguiente cuestionario.*

1. ¿Qué es una variable de tipo global?
2. ¿Qué es una variable de tipo local?
3. ¿Qué significa la palabra getch?
4. ¿Qué significa la palabra printf?
5. ¿Qué es un tipo definido por el usuario?
6. ¿Qué significa la palabra scanf?
7. ¿Qué es una variable?
8. ¿Qué es una constante?
9. Describe los operadores lógicos.
10. ¿Qué es una conversión de tipos?



# EXÁMENES DE AUTOEVALUACIÓN

I. De acuerdo con lo estudiado en estos temas, lee con atención las siguientes palabras y escribe cada una en la columna que le corresponda.

Palabras reservadas	Identificadores
1.	5.
2.	6.
3.	7.
4.	8.

a. extern	e. suma_1
b. lf	f. while
c. _t	g. float
d. y2	h. largo



*II. Elige la respuesta que consideres correcta:*

1. Un tipo de dato:

- a) Permite usar varios elementos en una estructura
- b) Determina las clases por usar
- c) Determina los objetos por usar
- d) Permite usar un rango de datos

2. El tamaño del tipo char es de:

- a) 1 byte
- b) 2 bytes
- c) 4 bytes
- d) 6 bytes

3. El tipo doublé se emplea para describir:

- a) números enteros
- b) letra o dígito
- c) números con decimales
- d) números y letras

4. La siguiente es una palabra reservada:

- a) var
- b) not
- c) If
- d) and



5. Una de las características de un identificador es que:
- a) Deben de comenzar por letra o por “\_”
  - b) Son secuencias que se forman empleando solo letras
  - c) Son símbolos cuyo significado está predefinido
  - d) Permite nuevos nombres para tipo de datos
6. Un ejemplo de operador relacional es:
- a) >
  - b) AND
  - c) &&
  - d) %
7. Las variables locales se declaran:
- a) dentro de una función
  - b) después de main ()
  - c) fuera de una función
  - d) antes de main ()
8. El nombre de una variable no debe sobrepasar los:
- a) 20 caracteres
  - b) 30 caracteres
  - c) 31 caracteres
  - d) 32 caracteres
9. Las variables que se almacenan en registros del microprocesador se denominan:
- a) locales
  - b) globales
  - c) externas



d) de registro

10. Son variables que se declaran dentro de una función:

- a) locales
- b) globales
- c) externas
- d) de registro

11. Para definir nuevos tipos de datos se utiliza la palabra reservada:

- a) register
- b) struct
- c) casting
- d) typedef

12. Una constante:

- a) es sinónimo de una variable
- b) tiene ámbito local
- c) tiene ámbito global
- d) mantiene su valor durante la ejecución del programa

13. La siguiente es una palabra reservada en C:

- a) echo
- b) println
- c) if
- d) writeln



14. Un ejemplo de un operador lógico es:

- a) >
- b) AND
- c) &&
- d) %

15. El operador módulo se escribe así:

- a) ++
- b) --
- c) %
- d) ||

16. Las variables globales se declaran:

- a) dentro de una función
- b) después de las constantes
- c) fuera de una función
- d) antes de main()

17. Un identificador puede empezar con:

- a) un número
- b) un espacio
- c) una letra
- d) un tabulador

18. El operador de negación es:

- a) ||
- b) &&
- c) %
- d) !



SUAYED

19. Es ejemplo de una biblioteca:

- a) stdio.h
- b) include
- c) define
- d) const

20. De los siguientes, el tipo de dato más grande es:

- a) int
- b) char
- c) float
- d) long int

21. Palabra reservada para devolver un valor:

- a) #include
- b) #define
- c) stdio.h
- d) return()



III. Observa las palabras/signos, y en base a lo que vimos en esta unidad, subraya las palabras /signos de la siguiente forma:

**ROJO** - las que correspondan a operadores aritméticos

**AZUL** - las que correspondan a asignaciones

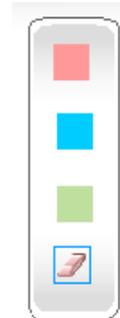
**VERDE** - las que correspondan a operadores relacionales y lógicos

a=2\*(++c), ++

variable=variable1;

>= +

%



Al finalizar la evaluación haz una impresión de pantalla, guárdala en tu computadora y una vez concluida, presiona el botón **Examinar**. Localiza el archivo ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.



## LO QUE APRENDÍ

Elabora un programa en donde utilices, por lo menos, una de las expresiones aritméticas (lógicas y relacionales), a la vez que cambies la prioridad de los operadores aritméticos utilizando paréntesis.



# MESOGRAFÍA

## Bibliografía sugerida

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

## Bibliografía básica

Cairó, O. (2003). *Metodología de la Programación*. (2ª ed.) México: Alfaomega.

Ceballos, F.J. (1997). *Lenguaje C*. 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*. (3ª ed.) Madrid: Pearson / Prentice Hall.

## Bibliografía complementaria

García, L.; Cuadrado, J.; De Amescua, A. y Velasco, M. (2004) *Construcción lógica de programas, Teoría y problemas resueltos*. México: Alfa Omega / Ra-Ma.

Joyanes Aguilar, Luis. (2005). *Programación en C++: Algoritmos, Estructuras de Datos y objetos*. (2ª ed.) Madrid: McGraw-Hill.

López, L. (2004). *Programación estructurada en turbo pascal 7*. México: Alfa Omega.

López, L. (2004a). *Programación estructurada, un enfoque algorítmico*. (2ª ed.) México: Alfa Omega.

Sedgewick, R. (1995). *Algoritmos en C++*. México: Addison-Wesley Iberoamericana.

Weiss, Mark Allen. (2000) *Estructuras de datos en JAVA*. México: Addison Wesley.



## Sitios de Internet

Sitio	Descripción
<a href="http://webdelprofesor.ula.ve/ingenieria/amoret/pd1/clase5.pdf">http://webdelprofesor.ula.ve/ingenieria/amoret/pd1/clase5.pdf</a>	Documentos que trata sobre las expresiones aritméticas, lógicas y relacionales
<a href="http://mimosa.pntic.mec.es/~flarrosa/lengc.pdf">http://mimosa.pntic.mec.es/~flarrosa/lengc.pdf</a>	Apunte sobre la estructura general de un programa en C
<a href="http://ing.otalca.cl/~rgarrido/cursos/progl_bio_2009/clases/cap4.htm">http://ing.otalca.cl/~rgarrido/cursos/progl_bio_2009/clases/cap4.htm</a>	Introducción al lenguaje de programación C
<a href="http://msdn.microsoft.com/es-es/library/x53a06bb%28v=VS.80%29.aspx">http://msdn.microsoft.com/es-es/library/x53a06bb%28v=VS.80%29.aspx</a>	Palabras claves de C#



## ANEXO 3

### Ejercicios: operadores lógicos y relacionales

#### Ejercicio 1

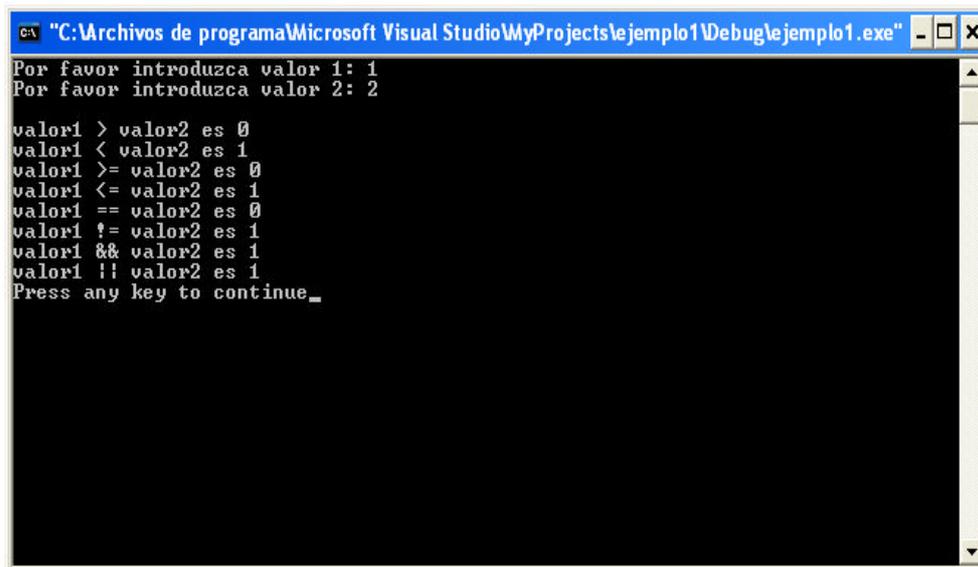
El siguiente ejercicio muestra el uso de los operadores lógicos y relacionales. El usuario debe introducir dos valores numéricos y el programa hace comparaciones relaciones y lógicas con los valores introducidos.

```
/*Este programa en C muestra el uso de los operadores  
lógicos y relacionales */  
#include <stdio.h>  
main()  
{  
    float valor1, valor2;  
  
    printf("Por favor introduzca valor 1: ");  
    scanf("%f",&valor1);  
    printf("Por favor introduzca valor 2: ");  
    scanf("%f",&valor2);  
    printf("\n");  
    printf("valor1 > valor2 es %d\n", (valor1>valor2));  
    printf("valor1 < valor2 es %d\n", (valor1<valor2));  
    printf("valor1 >= valor2 es %d\n", (valor1>=valor2));  
    printf("valor1 <= valor2 es %d\n", (valor1<=valor2));  
    printf("valor1 == valor2 es %d\n", (valor1==valor2));  
    printf("valor1 != valor2 es %d\n", (valor1!=valor2));  
    printf("valor1 && valor2 es %d\n", (valor1&&valor2));
```



```
printf("valor1 || valor2 es %d\n", (valor1||valor2));  
return(0);  
}
```

El resultado de este programa es el siguiente:



Veamos un programa que indica el menor de dos números leídos.



## EJERCICIO 2

```
/* Indica el menor de dos enteros leídos */
#include <stdio.h>
void main ( )
{
    int n1, n2, menor (int, int);
    printf ("Introducir dos enteros:\n");
    scanf ("%d%d", &n1, &n2);
    if ( n1 == n2 )
        printf ("Son iguales \n");
    else
        printf ("El menor es: %d\n",menor(n1,n2));
}
int menor (int a, int b)
{
    if ( a < b )
        return ( a );
    else
        return ( b );
}
```

El resultado del programa es el siguiente:



```
C:\Archivos de programa\Microsoft Visual Studio\MyProjects\ejemplo1\Debug\ejemplo1.exe
Introducir dos enteros:
1
2
El menor es: 1
Press any key to continue_
```



SUAYED  
UNA OPCIÓN  
PARA TI

Licenciatura: Informática

## UNIDAD 3

# CONTROL DE FLUJO

# APUNTES DIGITALES PLAN 2012



SUAYED  
UNA OPCIÓN  
PARA TI



## OBJETIVO ESPECÍFICO

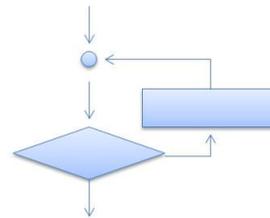
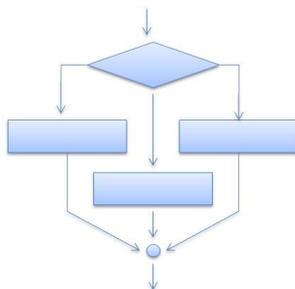
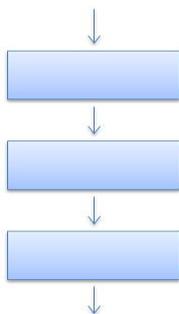
Al terminar la unidad, el alumno podrá utilizar las principales estructuras de la programación.

## 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:

- ♦ **Secuencial.** Ejecuta una sentencia después de otra.
- ♦ **Alternativa o Condicional.** Se evalúa una expresión y, dependiendo del resultado, se decide la siguiente sentencia por ejecutar.
- ♦ **Repetitiva o Iterativa.** Repetimos un bloque de sentencias hasta que sea verdadera una determinada condición. (Albert, Barber y Ferrís, 2004, p.39)





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.



## LO QUE SÉ

Explica qué es la programación estructurada y define cada uno de sus elementos.



# TEMARIO DETALLADO

**(14 horas)**

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

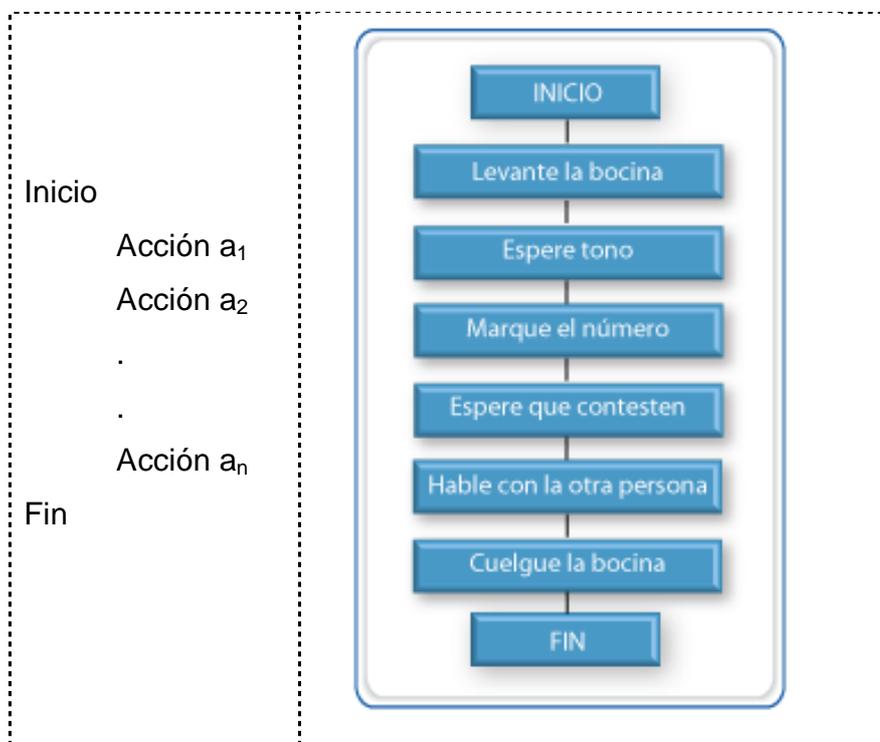


### 3.1. Estructura secuencial

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.

La estructura secuencial ejecuta las acciones sucesivamente, sin posibilidad de omitir ninguna y sin bifurcaciones. Todas estas estructuras tendrán una entrada y una salida.

*Ejemplo de un diagrama de flujo*





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

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<sup>2</sup> 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 de la línea del *if* y si es falsa, se ejecutan las

<sup>2</sup> 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. Para ahondar en este término, remitirse a: <http://msdn.microsoft.com/es-es/library/wts33hb3%28VS.80%29.aspx>



sentencias que se encuentran después de *else*, como se muestra a continuación:

```
if (expresion-booleana)
    Sentencia1;
else
    Sentencia2;
```

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.

<pre>switch (expresión){     case valor1:         sentencia;         break;     case valor2:         sentencia;         break;     case valor3:         sentencia;         break;     case valorN:         sentencia;         break;     default:  }</pre>	<pre>if (expresion-booleana) {     Sentencia1;     Sentencia2; } Else     Sentencia3;</pre>
--	---

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.ALTRAS\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;
```



Continúa Ej. 4

```
    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

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.

<b>While</b>	Ejecuta repetidamente el mismo bloque de código hasta que se cumpla una condición de terminación. Hay dos partes en un ciclo: <i>cuerpo</i> y <i>condición</i> .
<b>Cuerpo</b>	Es la sentencia o sentencias que se ejecutarán dentro del ciclo.
<b>Condición</b>	Es la condición de terminación del ciclo.

```
while(condición){  
    cuerpo;  
}
```

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.

```
do{
    cuerpo;
}while(terminación);
```



### 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");
```



SUAYED

```
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));
```



### Continúa Ej. 3

```
%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.

```
for (inicio; condición; incremento/decremento)
{
    sentencia(s);
}
```

### *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("Introduzca 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 DE LA UNIDAD

El teorema del programa estructurado indica que un programa puede escribirse empleando únicamente tres estructuras de control: secuencial (las sentencias se ejecutan en forma consecutiva), alternativa (dependiendo de la evaluación de una expresión se ejecuta una sentencia u otra) y repetitiva (repetición de un bloque de sentencias).

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:

- *Simple*: determina que se ejecute un fragmento de código basado en una condición. If (condición) then acción(es) else (la cláusula else es opcional), y,
- *Múltiple*: realiza distintas operaciones basadas en la evaluación de una variable o expresión. En el lenguaje C se ejecuta con switch (expresión).



En la estructura repetitiva la ejecución de las acciones dependen 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.



## GLOSARIO DE LA UNIDAD

### **Booleano**

El tipo de dato booleano es aquel que puede representar valores de lógica binaria, esto es, valores que representen "falso" o "verdadero".

### **Estructura secuencial**

En la estructura secuencial, todas las sentencias del programa se ejecutan sin posibilidad de omitir alguna.

### **Estructura alternativa**

En la estructura alternativa o selectiva, la ejecución del programa depende del cumplimiento de una condición o condiciones.

### **Estructura repetitiva**

En la estructura repetitiva, las sentencias se ejecutan un número determinado o indeterminado de veces.

### **Programación estructurada**

Paradigma de programación que divide un problema complejo en problemas más pequeños y fáciles de resolver.



## ACTIVIDADES DE APRENDIZAJE

### ACTIVIDAD 1

Elabora un programa en C que obtenga la edad de una persona utilizando como base, la fecha de nacimiento.

### ACTIVIDAD 2

Elabora un programa que determine la mensualidad que debe pagar una persona si pide un préstamo de \$10,000.00 pesos, tomando en cuenta una tasa de interés de 50% anual.

### ACTIVIDAD 3

Escribe un programa en lenguaje C que sume los números pares e impares dentro del rango del 1 al 100.

### ACTIVIDAD 4

Escribe un programa que valide una contraseña ingresada por el usuario. El usuario tendrá tres intentos para ingresar la contraseña correcta, de lo contrario, el programa mostrará el mensaje “usuario no autorizado”.



### ACTIVIDAD 5

Elabora un programa que obtenga la suma de los números ingresados por el usuario. Cuando el usuario ingrese un número cero, el programa se detendrá y mostrará el acumulado de la suma.

### ACTIVIDAD 6

Escribe un programa que obtenga la potencia de una base y un exponente dados, ocupando únicamente la operación de suma; no se deberá utilizar la multiplicación o la función de potencia.



# CUESTIONARIO DE REFORZAMIENTO

Contesta el siguiente cuestionario.

1. Define qué es una estructura secuencial.
2. Define qué es una estructura alterativa.
3. Explica qué es la estructura repetitiva.
4. Es un ciclo con un número determinado de iteraciones. Entonces nos estamos refiriendo a:
5. Es un ciclo con un número indeterminado de iteraciones. Hacemos referencia a:
6. Es un ciclo que se ejecuta al menos una vez, independientemente de que se cumpla o no una condición. Entonces estamos hablando de:
7. ¿Cuál es la función de la instrucción *default*?
8. ¿Cuál es la función de la instrucción *switch*?
9. ¿Cuál es la función de la palabra reservada *case*?
10. ¿Cuál es la función de la palabra reservada *break*?



## EXÁMENES DE AUTOEVALUACIÓN

I. Selecciona si las aseveraciones son verdaderas o falsas.

	Verdadera	Falsa
1. La estructura alternativa permite que el programa fluya de acuerdo a una condición.	( )	( )
2. La estructura alternativa utiliza la palabra reservada <i>while</i> .	( )	( )
3. En la estructura alternativa <i>if-then</i> , si la condición no se cumple se puede utilizar la palabra <i>else</i> para cambiar el flujo.	( )	( )
4. En lugar de utilizar varios <i>if</i> , una opción es emplear la palabra reservada <i>case</i> .	( )	( )
5. Dentro del <i>case</i> , se evalúa la opción.	( )	( )



*II. Lee cada una de las expresiones y elige la respuesta correcta.*

1. Tipo de estructura que permite que todas las sentencias se ejecuten sin posibilidad de omitir alguna:

- a) Secuencial
- b) Alternativa
- c) Selectiva
- d) Repetitiva

2. Tipo de estructura que permite que el flujo de un programa se bifurque:

- a) Alternativa
- b) Selectiva
- c) If
- d) Else

3. Tipo de estructura que permite que una sentencia se ejecute un número determinado de veces:

- a) do
- b) while
- c) for
- d) main ()

4. Tipo de estructura que permite que una sentencia se ejecute un número indeterminado de veces:

- a) default
- b) while
- c) for
- d) else



5. Es un ciclo que permite que se ejecute una sentencia, al menos una vez, independientemente de que se cumpla o no una condición:

- a) ciclo
- b) while
- c) do-while
- d) for

6. Un ciclo de tipo for es un ciclo:

- a) con un número determinado de iteraciones
- b) con un número indeterminado de iteraciones
- c) infinito
- d) de tipo do-while

7. *break* es usado para:

- a) Salir de un ciclo
- b) Entrar a un ciclo
- c) Determinar el flujo de un programa
- d) Crear una función

8. La siguiente es una palabra reservada que se usa en la función *switch* en caso de no se cumpla ningún caso:

- a) break
- b) default
- c) while
- d) else



SUAYED

9. Es una palabra reservada que permite que el programa se ejecute de acuerdo al cumplimiento de una condición:

- a) if
- b) where
- c) case
- d) for

10. Es una función que permite la selección de varias opciones:

- a) include
- b) switch
- c) break
- d) for



## LO QUE APRENDÍ

Elabora un programa que, a través del siguiente menú, realice las operaciones de un cajero automático:

Menú Principal

- |              |             |
|--------------|-------------|
| 1. Depósitos | 3. Consulta |
| 2. Retiros   | 4. Salir    |

Restricciones:

→ El programa seguirá ejecutándose en tanto el usuario no presione la opción 4, de Salir.

→ El usuario:

- ♦ Podrá realizar el número de depósitos que desee por cualquier cantidad.
- ♦ No podrá realizar un retiro por una cantidad mayor al saldo de la cuenta.
- ♦ Podrá consultar en cualquier momento el saldo de su cuenta.

→ Utiliza las estructuras de control aprendidas en esta unidad.



# MESOGRAFÍA

## Bibliografía sugerida

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

## Bibliografía básica

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

Cairó, O. (2003). *Metodología de la Programación*. (2ª ed.) México: Alfaomega.

Ceballos, F.J. (1997). *Lenguaje C*. México: Alfaomega.

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



SUAYED

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

## Bibliografía complementaria

Albert Blanco, Jesús V.; Barber Miralles, Fernando. y Ferrís Castell, Ricardo. (2004) “4. Programación estructurada”, *Algoritmos y estructuras de datos I*. Universidad de Valencia, Departamento de Informática. Disponible en línea:

[http://informatica.uv.es/iiguia/AED/oldwww/2004\\_05/AED.Tema.04.pdf](http://informatica.uv.es/iiguia/AED/oldwww/2004_05/AED.Tema.04.pdf), consultado el 20/09/11.

López, L. (2004) *Programación estructurada. Un enfoque algorítmico* (2ª ed.) México: Alfa Omega.

Sedgewick, R. (1995) *Algoritmos en C++*. México: Addison-Wesley Iberoamericana.

Weiss, M. A. (2000) *Estructuras de datos en JAVA*. México: Addison Wesley.



## Sitios de Internet

Sitio	Descripción
<a href="http://www.monografias.com/trabajos/progestructu/progestructu.shtml">http://www.monografias.com/trabajos/progestructu/progestructu.shtml</a>	Artículo que trata sobre la programación estructurada.
<a href="http://www.itescam.edu.mx/principales/sylabus/fpdb/recursos/r51758.PDF">http://www.itescam.edu.mx/principales/sylabus/fpdb/recursos/r51758.PDF</a>	Técnicas de programación estructurada
<a href="http://sistemas.itlp.edu.mx/tutoriales/pascal/u1_1_4.html">http://sistemas.itlp.edu.mx/tutoriales/pascal/u1_1_4.html</a>	Programación estructurada
<a href="http://msdn.microsoft.com/es-es/library/xt4z8b0f%28v=VS.80%29.aspx">http://msdn.microsoft.com/es-es/library/xt4z8b0f%28v=VS.80%29.aspx</a>	Tipos de instrucciones (Referencia de C#)
<a href="http://msdn.microsoft.com/es-es/library/x53a06bb%28v=VS.80%29.aspx">http://msdn.microsoft.com/es-es/library/x53a06bb%28v=VS.80%29.aspx</a>	Palabras claves de C#
<a href="http://correo.uan.edu.mx/~iavalos/Estructuras%20de%20desicion.html">http://correo.uan.edu.mx/~iavalos/Estructuras%20de%20desicion.html</a>	Estructura alternativa/de decisión/de selección



SUAYED  
UNA OPCIÓN  
PARA TI

Licenciatura: Informática

---

## UNIDAD 4

## FUNCIONES

# APUNTES DIGITALES PLAN 2012



SUAYED  
UNA OPCIÓN  
PARA TI



## OBJETIVO ESPECÍFICO

Al terminar la unidad, el alumno utilizará las funciones preconstruidas y podrá desarrollar sus propias funciones; identificará el alcance de las variables utilizadas y aplicará la recursividad.



## INTRODUCCIÓN

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

Las declaraciones de funciones generalmente son especificadas por:

- 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:

```
int mayor (int num1, int num2)
{
    if (num1>num2)
        return num1;
    else
        return num2;
}
```



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.



## LO QUE SÉ

Con base en lo que hemos visto en las unidades anteriores y por lo que conoces de este tema, intenta realizar una función en pseudocódigo que determine el mayor de dos números introducidos por el usuario. Cualquier duda pregúntala a tu asesor.



# TEMARIO DETALLADO

**(18 horas)**

## 4.1. Internas

4.1.1. Funciones de caracteres y cadenas

4.1.2. Funciones matemáticas

4.1.3. Funciones de conversión

## 4.2. Definidas por el usuario

4.3. Ámbito de variables (locales y globales)

4.4. Recursividad

## 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<sup>3</sup> del lenguaje C, como por ejemplo: el archivo `string.h` contiene las funciones que se utilizan para el manejo de cadenas de caracteres; `math.h` define las funciones matemáticas utilizadas en C y C++; `stdlib.h` contiene las funciones de conversión de tipos, de búsqueda y ordenación, etcétera.

### 4.1.1 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.

---

<sup>3</sup> 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. <b>int</b> DISTINTO DE CERO si la <b>isalpha(variable_char);</b> variable es una letra del alfabeto, en caso contrario devuelve cero. Cabecera: <ctype.h>.
isdigit	Devuelve un entero. <b>int</b> DISTINTO DE CERO si la <b>isdigit(variable_char);</b> variable es un número (0 a 9), en caso contrario devuelve cero. Cabecera <ctype.h>.
isgraph	Devuelve un entero. <b>int</b> DISTINTO DE CERO si la <b>isgraph(variable_char);</b> variable es cualquier carácter imprimible distinto de un espacio, si es un espacio CERO. Cabecera <ctype.h>.
islower	Devuelve un entero. <b>int</b> DISTINTO DE CERO si la <b>islower(variable_char);</b> variable está en minúscula, en caso contrario devuelve cero. Cabecera <ctype.h>.
ispunct	Devuelve un entero. <b>int</b> DISTINTO DE CERO si la <b>ispunct(variable_char);</b> variable es un carácter de puntuación, en caso contrario, devuelve cero.



Cabecera <ctype.h>	
isupper	Devuelve un entero. <b>int</b>
	DISTINTO DE CERO si la <b>isupper(variable_char);</b> variable está en mayúsculas, en caso contrario, devuelve cero. Cabecera <ctype.h>

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		
<code>memset</code>	Inicializar una región de memoria (buffer) con un valor determinado. Se utiliza principalmente para inicializar cadenas con un valor determinado. Cabecera <code>&lt;string.h&gt;</code> .	<code>memset</code> <code>(var_cadena,'carácter',tamaño);</code>
<code>strcat</code>	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 <code>&lt;string.h&gt;</code> .	<code>strcat(cadena1,cadena2);</code>



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>.	<pre>strchr(cadena,'carácter');</pre> <pre>strchr("texto",'carácter');</pre>										
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>.	<pre>strcmp(cadena1,cadena2);</pre> <pre>strcmp(cadena2,"texto");</pre> <table border="1" data-bbox="791 1043 1107 1335"> <thead> <tr> <th colspan="2">RESULTADO</th> </tr> <tr> <th>VALOR</th> <th>DESCRIPCIÓN</th> </tr> </thead> <tbody> <tr> <td>Menor a</td> <td>Cadena1 menor a Cadena2.</td> </tr> <tr> <td>Cero</td> <td>Las cadenas son iguales.</td> </tr> <tr> <td>Mayor a</td> <td>Cadena1 mayor a Cadena2.</td> </tr> </tbody> </table>	RESULTADO		VALOR	DESCRIPCIÓN	Menor a	Cadena1 menor a Cadena2.	Cero	Las cadenas son iguales.	Mayor a	Cadena1 mayor a Cadena2.
RESULTADO												
VALOR	DESCRIPCIÓN											
Menor a	Cadena1 menor a Cadena2.											
Cero	Las cadenas son iguales.											
Mayor a	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	<pre>strcpy(cadena1,cadena2);</pre> <pre>strcpy(cadena1,"texto");</pre>										



	contemplar es que el tamaño de la segunda cadena sea menor o igual al tamaño de la primera cadena. Cabecera <string.h>.	
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>	<code>variable=strlen(cadena);</code>
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>.	<code>strncat(cadena1,cadena2,nº de caracteres);</code>
strncmp	Compara	<code>strncmp(cadena1,cadena2,nº de</code>



	<p>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 &lt;string.h&gt;.</p>	<pre>caracteres); strncmp(cadena1,"texto",nº de caracteres);</pre>
strncpy	<p>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 &lt;string.h&gt;.</p>	<pre>strncpy(cadena1,cadena2,nº de caracteres); strncpy(cadena1,"texto",nº de caracteres);</pre>
strrchr	<p>Devuelve un puntero a la última ocurrencia del carácter buscado en la cadena. Si no lo encuentra devuelve un puntero nulo. Cabecera &lt;string.h&gt;.</p>	<pre>strrchr(cadena,'carácter'); strrchr("texto",'carácter');</pre>
strpbrk	<p>Devuelve un puntero al primer carácter de la</p>	<pre>strpbrk("texto","cadena_de_busqueda ");</pre>



	cadena que coincida con otro de la cadena a buscar. Si no hay correspondencia devuelve un puntero nulo. Cabecera <string.h>.	<code>strpbrk(cadena,cadena_de_busqueda);</code>
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>.	<code>variable_char=tolower(variable_char);</code>
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>.	<code>variable_char=toupper(variable_char);</code>

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();
}
```



### 4.1.2. 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	Devuelve un tipo <i>double</i> . Muestra el arcocoseno de la variable, ésta debe ser de tipo <i>double</i> y estar en el rango -1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>.	<i>double</i> acos(variable_double);
asin	Devuelve un tipo <i>double</i> . Muestra el arcoseno de la variable, ésta debe ser de tipo <i>double</i> y estar en el rango -1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>.	<i>double</i> asin(variable_double);
atan	Devuelve un tipo <i>double</i> . Muestra el arcotangente de la variable, ésta debe ser de tipo <i>double</i> y estar en el rango -1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>.	<i>double</i> atan(variable_double);
cos	Devuelve un tipo <i>double</i> . Muestra el coseno de la variable, ésta debe ser de tipo <i>double</i> y estar expresada en radianes. Cabecera <math.h>.	<i>double</i> cos(variable_double_radianes);
sin	Devuelve un tipo <i>double</i> . Muestra el seno de la variable, ésta debe ser de	<i>Double</i> sin(variable_double)



	tipo <i>double</i> y estar expresada en radianes. Cabecera <math.h>.	<code>e_radianes);</code>
tan	Devuelve un tipo <i>double</i> . Muestra la tangente de la variable, ésta debe ser de tipo <i>double</i> y estar expresada en radianes. Cabecera <math.h>.	<code>double tan(variable_double_radianes);</code>
cosh	Devuelve un tipo <i>double</i> . Muestra el coseno hiperbólico de la variable, ésta debe ser de tipo <i>double</i> y estar en el rango -1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.	<code>double cosh(variable_double);</code>
sinh	Devuelve un tipo <i>double</i> . Muestra el seno hiperbólico de la variable, ésta debe ser de tipo <i>double</i> y estar en el rango -1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.	<code>double sinh(variable_double);</code>
tanh	Devuelve un tipo <i>double</i> . Muestra la tangente hiperbólica de la variable, ésta debe ser de tipo <i>double</i> y estar en el rango -1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.	<code>double tanh(variable_double);</code>
ceil	Devuelve un <i>double</i> 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,	<code>double ceil(variable_double);</code>



	redondea la variable a la alta. Cabecera <math.h>.	
floor	Devuelve un <i>double</i> 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>.	<code>double floor(variable_double);</code>
fabs	Devuelve un valor <i>float</i> o <i>double</i> . Devuelve el valor absoluto de una variable <i>float</i> . Se considera una función matemática, pero su cabecera es <stdlib.h>.	<code>var_float fabs(variable_float) ;</code>
labs	Devuelve un valor <i>long</i> . Devuelve el valor absoluto de una variable long. Se considera una función matemática, pero su cabecera es <stdlib.h>.	<code>var_long labs(variable_long) ;</code>
abs	Devuelve un valor entero. Devuelve el valor absoluto de una variable int. Se considera una función matemática, pero su cabecera es <stdlib.h>.	<code>var_float abs(variable_float);</code>
modf	Devuelve un <i>double</i> . 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	<code>var_double_decimal= modf(variable,var_parte_entera);</code>



	<i>double</i> . Cabecera <math.h>.	
pow	Devuelve un <i>double</i> . 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>.	<code>var_double=pow(b ase_double,expon ente_double);</code>
sqrt	Devuelve un <i>double</i> . 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>.	<code>var_double=sqrt(v ariable_double);</code>
log	Devuelve un <i>double</i> . 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>.	<code>double log(variable_doubl e);</code>
log10	Devuelve un valor <i>double</i> . Realiza el logaritmo decimal de la variable de tipo <i>double</i> . 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>.	<code>double log10(var_double);</code>
randomize()	Inicializa la semilla para generar números aleatorios. Utiliza las funciones de tiempo para crear esa semilla. Esta función está relacionada	<code>void randomize();</code>



con random. Cabecera <stdlib.h>.

random

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_i  
nt);
```

### 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.



### 4.1.3 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>.	<pre>int atoi(variable_char) ;</pre>
atol	Devuelve un <i>long</i> . Esta función convierte la cadena en un valor <i>long</i> . 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>.	<pre>long atol(variable_char) ;</pre>
atof	Devuelve un <i>double</i> . Esta función convierte la cadena en un valor <i>double</i> . La cadena debe contener un número <i>double</i> válido, si no es así, el valor devuelto queda indefinido. La cadena puede terminar con espacios en	<pre>double atof(variable_char) ;</pre>



blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es <stdlib.h>.

**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.



Itoa	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>.	<code>ltoa(var_long,var_cadena,base);</code>
------	--	--

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.

**SINTAXIS DEL** `tipo_devuelto nombre_funcion ([parámetros]);`

**PROTOTIPO:**

**SINTAXIS DE LA** `nombre_funcion([parámetros]);`

**LLAMADA:**

**SINTAXIS DEL** `tipo_devuelto nombre_funcion ([parámetros])`

**DESARROLLO:**

```
{  
    cuerpo;  
}
```



### 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 raíz 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)

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.

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

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:

respuest a	respuesta*factorial(respues ta-1.0)	retur n
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 DE LA UNIDAD

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. Las internas son las funciones estándar que contiene el propio lenguaje, como por ejemplo: funciones para el manejo de caracteres y cadenas, matemáticas, de conversión, etcétera, las cuales están almacenadas dentro de las bibliotecas que se encuentran en archivos de cabecera (por ejemplo el archivo string.h contiene las funciones de caracteres y cadena). 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: por **valor** o por **referencia**. En la primera se copia el contenido del argumento al parámetro de la función, de modo que los cambios que realice la función sobre sus parámetros no afectan a los argumentos; por referencia se pasa a la función la dirección de memoria del argumento, si se producen cambios en el parámetro de la función, afectan también al argumento.



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**, aquellas que se declaran dentro de una función y su ámbito se restringe dentro de la misma función, y **variables globales**, cuyos valores son conocidos en todo el programa y se declaran fuera de todas las funciones.

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.



SUAYED  
Sistema Universitario  
Autónomo de Uruguay

## GLOSARIO DE LA UNIDAD

### **Función**

Segmento de código de un programa que realiza una tarea específica.

### **Función definida por el usuario**

Función creada por un programador para un fin específico.

### **Función interna**

Función predefinida en un lenguaje de programación.

### **Recursividad**

Es la capacidad de una función de llamarse a sí misma.

### **Variable global**

Variable cuyo ámbito es global a todo el programa, en otras palabras, dicha variable se puede usar en cualquier parte del programa.

### **Variable local**

Variable cuyo ámbito se restringe a la función donde fue creada.



## ACTIVIDADES DE APRENDIZAJE

### ACTIVIDAD 1

Elabora un programa que, a través de una función, transforme una cadena introducida por el usuario en minúsculas, a su equivalente en mayúsculas.

### ACTIVIDAD 2

Elabora un programa en C que acepte el ingreso de tres números y que, por medio de una función, los ordene en forma ascendente.

### ACTIVIDAD 3

Elabora una función que busque un número que introduzca el usuario, en un arreglo de 100 números.

### ACTIVIDAD 4

Elabora una función que determine los factores de un número introducido por el usuario; por ejemplo: si se ingresa el número 8, los factores del número serán 2, 2, 2 y 1.



### ACTIVIDAD 5

Escribe una función que obtenga las combinaciones de tres enteros introducidos por el usuario. Ejemplo: si se ingresan los números 1, 3, y 5, se imprimirán las combinaciones 1,3,5; 1,5,3; 3,1,5; 3,5,1; 5,1,3 y 5,3,1.

### ACTIVIDAD 6

Elabora una función que calcule el pago de mensualidades para una deuda de \$16,000.00, tomando en cuenta un plazo de un año, y un interés anual de 24 %; utiliza variables globales.

### ACTIVIDAD 7

Elabora una función que obtenga la factorial de un número. Ejemplo: si se ingresa el 4, el resultado es 24, porque  $4 \times 3 \times 2 \times 1 = 24$



SUAYED  
Sistema Universitario  
Autónomo de Uruguay

# CUESTIONARIO DE REFORZAMIENTO

*Contesta el siguiente cuestionario.*

1. ¿Qué es una función?
2. ¿Qué es la recursividad?
3. ¿Qué significa la palabra void?
4. ¿Qué es un parámetro?
5. ¿Qué significa main?
6. ¿Qué es un parámetro por valor?
7. ¿Qué es un parámetro por referencia?
8. ¿Qué es una función desarrollada por el usuario?
9. ¿Cuál es la función de return?
10. ¿Qué es una función interna?



## EXAMEN DE AUTOEVALUACIÓN

*Lee con atención las siguientes frases y selecciona la respuesta correcta.*

1. Las funciones isalpha se encuentra en la librería:

- a) math.h
- b) string.h
- c) stdio.h
- d) ctype.h

2. Las funciones strcat se encuentran en la librería:

- a) math.h
- b) string.h
- c) stdio.h
- d) ctype.h

3. Es una función que transforma una cadena en un valor entero:

- a) atol
- b) atoi
- c) itoa
- d) sprintf

4. La función randomize se encuentra en la librería:

- a) math.h
- b) string.h



SUAVEO

- c) stdio.h
- d) ctype.h

5. Es una función que transforma un número a su equivalente en cadena:

- a) atol
- b) atoi
- c) itoa
- d) sprintf

6. Cuando se copia el contenido del argumento al parámetro de la función se hace un paso de parámetros por:

- a) valor
- b) referencia
- c) apuntadores
- d) variables

7. Para devolver un valor se usa la palabra reservada:

- a) void
- b) sizeof
- c) int
- d) return

8. La palabra reservada void, antes de una función, indica que la función:

- a) devuelve un valor
- b) no devuelve nada
- c) devuelve un entero
- d) devuelve un tipo char



9. Para desarrollar una función el primer paso es:
- a) definir los parámetros
  - b) definir las variables
  - c) llamar a la función
  - d) crear el prototipo
10. La palabra reservada `void` en los parámetros indica que:
- a) la función carece de parámetros
  - b) la función tiene un parámetro
  - c) la función tiene más de un parámetro
  - d) no devuelve valor alguno
11. La palabra `int` antes de una función indica que:
- a) La función es de tipo entero
  - b) La función devuelve un entero
  - c) Sus parámetros son enteros
  - d) Que la función no devuelve valores
12. El parámetro de una función puede ser:
- a) un tipo de dato
  - b) otra función
  - c) una biblioteca
  - d) variables o constantes
13. Las variables de tipo `register`:
- a) son variables globales
  - b) se almacenan en registros del microprocesador
  - c) son variables locales
  - d) son variables externas



14. La función *getch()* se encuentra en la biblioteca:

- a) string.h
- b) stdio.h
- c) conio.h
- d) dos.h

15. La función *islower* devuelve un:

- a) entero
- b) flotante
- c) carácter
- d) long

16. Para inicializar una región de memoria se usa la función:

- a) scanf
- b) memset
- c) cin
- d) cout

17. La función *isalpha* se encuentra en la biblioteca:

- a) ctype.h
- b) string.h
- c) sodio.h
- d) conio.h

18. La función *sqrt* obtiene:

- a) la potencia de un número
- b) la raíz cuadrada de un número
- c) el logaritmo de un número



SUAYED

d) el coseno de un número

19. La biblioteca que contiene las funciones de conversión es:

- a) string
- b) conio
- c) stdlib
- d) math

20. Una función recursiva es una función que:

- a) se llama a sí misma
- b) devuelve valores
- c) no devuelve valores
- d) usa parámetros



## LO QUE APRENDÍ

Elabora un programa que muestre el siguiente menú:

1. Ingresar un número
2. Incrementar un número
3. Decrementar un número
4. Salir

Cada una de las opciones del menú llamará a una función que realizará una tarea específica, como se indica a continuación:

- La opción 1 aceptará el ingreso de un número entero; si no se ocupa esta opción, se iniciará con el número 10.
- La opción 2 le sumará un 1 al número anterior; y la opción 3 le restará un 1 al número mencionado. Utiliza variables globales.



# MESOGRAFÍA

## Bibliografía sugerida

Autor	Capítulo	Páginas
Joyanes (2006)	6	217-269

## Bibliografía básica

Ceballos, F.J. (1997). *Lenguaje C*. México: Alfaomega.

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

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

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



## Bibliografía complementaria

García, L.; Cuadrado, J.; De Amescua, A. y Velasco, M. (2004) *Construcción lógica de programas, Teoría y problemas resueltos*. México: Alfa Omega / Ra-Ma.

López, L. (2004). *Programación estructurada, un enfoque algorítmico*. (2ª. ed.) México: Alfa Omega.

Sedgewick, R. (1995). *Algoritmos en C++*. México: Addison-Wesley Iberoamericana.

## Sitios de Internet

Sitio	Descripción
<a href="http://c.conclase.net/curso/?cap=024">http://c.conclase.net/curso/?cap=024</a>	La función recursiva
<a href="http://www.data-2013.cl/DOCS/INFORMATICA/PROGRC/cap-c9.html">http://www.data-2013.cl/DOCS/INFORMATICA/PROGRC/cap-c9.html</a>	Bibliotecas de funciones en C
<a href="http://www.modelo.edu.mx/univ/virtech/prograc/cplus2.htm">http://www.modelo.edu.mx/univ/virtech/prograc/cplus2.htm</a>	Funciones en C++
<a href="http://www.aprender-informatica.com/teoria-de-la-programacion/cursos-de-informatica-teoria-de-la-programacion-estructuras-de-control.php">http://www.aprender-informatica.com/teoria-de-la-programacion/cursos-de-informatica-teoria-de-la-programacion-estructuras-de-control.php</a>	Teoría de la programación



SUAYED  
UNA OPCIÓN  
PARA TI

Licenciatura: Informática

## UNIDAD 5

# TIPOS DE DATOS COMPUESTOS (ESTRUCTURAS)

## APUNTES DIGITALES PLAN 2012



SUAYED  
UNA OPCIÓN  
PARA TI



## OBJETIVO ESPECÍFICO

Al terminar la unidad, el alumno podrá utilizar arreglos unidimensionales, multidimensionales y estructuras, para almacenar y procesar datos para aplicaciones específicas.



## 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.



SUAYED  
Sistema Universitario  
Autónomo de Uruguay

## LO QUE SÉ

Enlista cinco conjuntos de valores que consideres que se puedan almacenar en la memoria de una computadora, y que sirvan para realizar alguna tarea en específico.



# TEMARIO DETALLADO

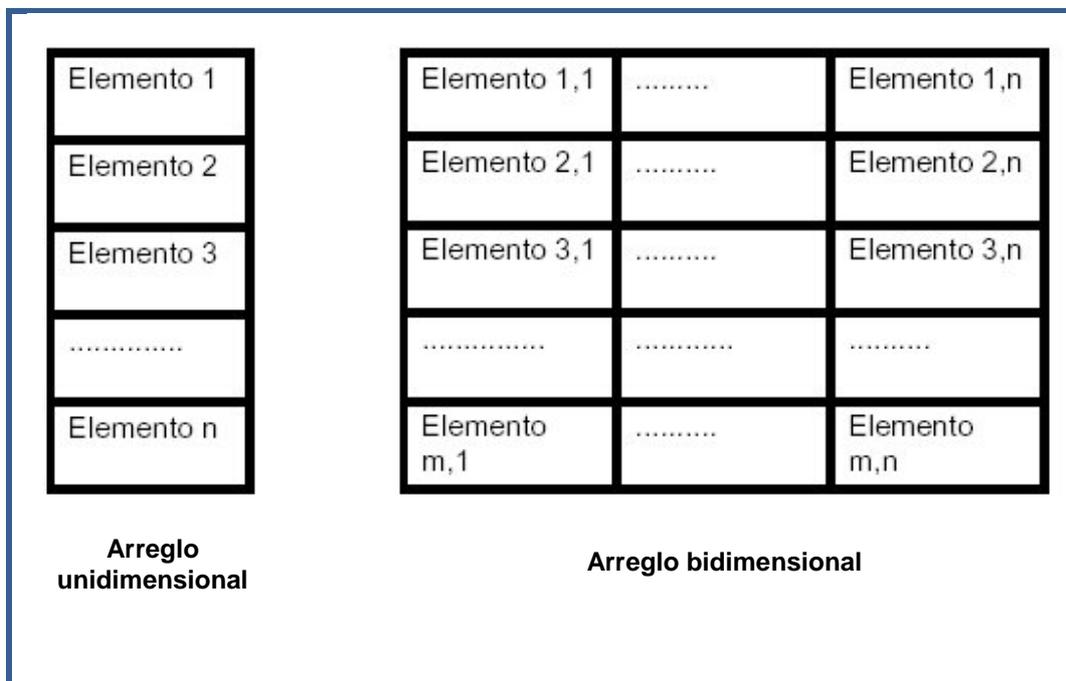
**(14 horas)**

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



## 5.1. Arreglos unidimensionales

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:

`tipo nom_ar [ tamaño ]`

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:

```
tipo nombre_arr [ tam1 ][ tam2 ] ... [ tamN];
```

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

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

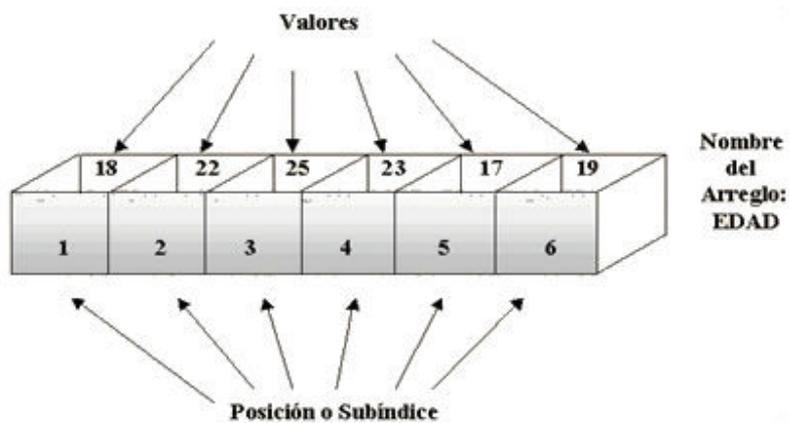


Observar que, para declarar cada dimensión, lleva sus propios corchetes.

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

```
tabladenum[2][3] = 15; /* Asigna 15 al elemento de la 3ª fila y la 4ª
                        columna*/
```

```
num = tabladenum[25][16];
```



### 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.

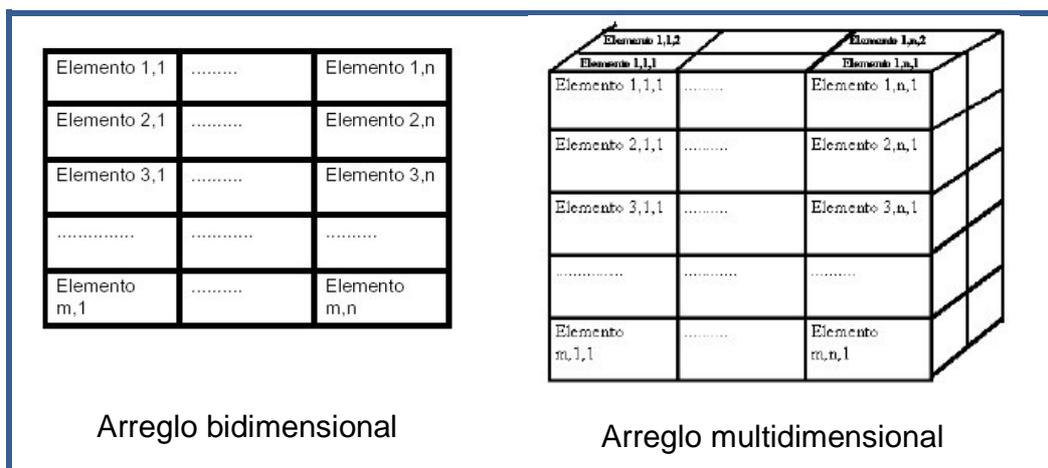
```
#include <stdio.h>
int main(void)
{
    int x[100]; /* declara un arreglo de 100-
integer */
    int t;
    /* carga x con valores de 0 hasta 99 */
    for(t=0; t<100; ++t)
x[t] = t;
    /* despliega el contenido de x */
    for(t=0; t<100; ++t)
printf("%d ", x[t]);
    return 0;
}
```



## 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.





## SINTAXIS

tipo nombre\_arreglo[fil][columna];

tipo nomb\_arreglo[fil][col]={ {v1,v2,vN},{v1,v2,vN},{vN} };

tipo nomb\_arreglo[][]={ {v1,v2,vN},{v1,v2,vN},{vN} };

num [fila] [columna]				
	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

### Inicialización de un elemento

nombre\_arreglo[indice\_fila][indice\_columna]=valor;

### Utilización de un elemento

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



## Ejemplos

### 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:

```
char cadena[11];
```

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:

```
char nombre_arr[ tam ]="cadena";
```

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.

Una **estructura** es una 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(); /*vacía la memoria intermedia del teclado*/

printf("\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);
    fflush(); /*vacía 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(); /*vacía la memoria intermedia del teclado*/

        printf("\n\n\n");
    }
}
```



```
printf("Un %s %s con %d años de antigüedad con número de
matricula
#%s\n",miauto.fabricante,miauto.modelo,miauto.antigüedad,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
```



SUAYED  
Sistema Universitario  
Autónomo de Valparaíso

d 4 600.000000  
0 0.000000  
0 0.000000



## RESUMEN DE LA UNIDAD

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 multidimensionales 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.



## GLOSARIO DE LA UNIDAD

### **Arreglo**

Conjunto de variables del mismo tipo.

### **Arreglo multidimensional**

Arreglo que consta de más de dos dimensiones.

### **Cadena**

Conjuntos de caracteres.

### **Estructura**

Conjunto de variables de distinto tipo.

### **Matriz**

Arreglo que consta de dos dimensiones, para acceder a un elemento dentro de una matriz se usan dos índices, una indica el renglón y otra la columna.



## ACTIVIDADES DE APRENDIZAJE

### ACTIVIDAD 1

Escribe un programa que determine la suma de los números almacenados en la diagonal de un arreglo de tamaño 4 x 4.

### ACTIVIDAD 2

Elabora un programa que determine si el contenido de dos arreglos, son iguales.

### ACTIVIDAD 3

Elabora un programa en C que multiplique dos matrices, ambas de tamaño 3 X 3.

### ACTIVIDAD 4

Haz un programa que cuente la cantidad de letras que contenga una frase, ingresada por el usuario, almacenada en un arreglo de caracteres.



### **ACTIVIDAD 5**

Elabora un programa que cuente la cantidad de palabras de una frase almacenada en un arreglo de caracteres.

### **ACTIVIDAD 6**

Elabora un programa para almacenar los datos de un alumno, dichos datos son: número de cuenta, nombre, licenciatura, semestre en curso y promedio. Dicha información será almacenada en una estructura de registro.



SUAYED  
SISTEMAS DE  
AYUDA PARA  
EL DESARROLLO

# CUESTIONARIO DE REFORZAMIENTO

*Contesta el siguiente cuestionario.*

1. ¿Qué es un arreglo?
2. ¿Qué es un arreglo unidimensional?
3. ¿Qué es un arreglo multidimensional?
4. ¿Qué es el índice de un arreglo?
5. ¿Qué es una estructura?
6. ¿Qué es un miembro de una estructura?
7. ¿Cuántos tipos de datos puede almacenar un arreglo?
8. ¿Cuántos tipos de datos puede almacenar una estructura?
9. ¿Qué es una cadena?
10. ¿Cuál es la utilidad de la función fflush()?



## EXAMEN DE AUTOEVALUACIÓN

*Lee con atención las siguientes frases y selecciona la respuesta correcta.*

1. Un arreglo es:

- a) Un grupo de elementos del mismo tipo
- b) Un grupo de elementos de tipos distintos
- c) Un tipo de dato
- d) Una variable

2. Un arreglo puede usar:

- a) Cualquier tipo de dato
- b) Solo cadenas
- c) Solo números
- d) Solo flotantes

3. Si el arreglo se inicializa con una cadena se usa:

- a) `
- b) “
- c) (
- d) {



4. El primer elemento de un arreglo se ubica en la posición:

- a) 0
- b) 1
- c) -1
- d) 2

5. Un arreglo unidimensional es sinónimo de un:

- a) grupo de elementos de distinto tipo
- b) vector
- c) vector de vectores
- d) arreglo con tres índices

6. Un arreglo multidimensional es un:

- a) grupo de elementos de distinto tipo
- b) vector
- c) arreglo de arreglos
- d) arreglo con tres índices

7. Para acceder a un arreglo bidimensional se usa:

- a) Un índice
- b) Un vector
- c) Dos índices
- d) Tres índices



8. Si se almacena un número introducido por el usuario en un arreglo este debe ser precedido por:

- a) &
- b) \*
- c) &&
- d) ->

9. Un arreglo bidimensional es sinónimo de:

- a) Vector
- b) Matriz
- c) Estructura
- d) Enumeración

10. Si un arreglo es declarado de tipo carácter, es suficiente declarar la biblioteca:

- a) stdio.h
- b) string.h
- c) conio.h
- d) stdlib.h

11. Para una cadena de 5 letras se deben asignar:

- a) 5 espacios en un arreglo
- b) 4 espacios en un arreglo
- c) 6 espacios en un arreglo
- d) 7 espacios en un arreglo



12. El tipo de dato para una cadena es:

- a) string
- b) char
- c) int
- d) float

13. Una cadena termina con el caracter:

- a) /n
- b) /t
- c) /0
- d) /s

14. Para inicializar un carácter se usa:

- a) #
- b) “
- c) ´
- d) (

15. Para inicializar una cadena se utiliza la palabra:

- a) int
- b) static
- c) float
- d) string

16. El primer elemento de un arreglo se indica con el número:

- a) 1
- b) 0
- c) -1
- d) NULL



17. Un arreglo unidimensional es sinónimo de:

- a) Un vector
- b) Una variable
- c) Registro
- d) Índice

18. Para acceder al contenido de un arreglo unidimensional se necesita:

- a) Usar un índice
- b) Usar dos índices
- c) Usar tres índices
- d) Usar cuatro índices

19. Un arreglo permite:

- a) Almacenar un tipo de dato
- b) Almacenar dos tipos de datos
- c) Almacenar tres tipos de datos
- d) Almacenar varios tipos de datos

20. Una estructura permite:

- a) Almacenar un tipo de dato
- b) Almacenar dos tipos de datos
- c) Almacenar tres tipos de datos
- d) Almacenar varios tipos de datos



21. Una cadena termina con el caracter:

- a) Retorno de carro
- b) Tabulador
- c) Nueva línea
- d) NULO

22. Para inicializar una cadena en un arreglo se usa:

- a) La comilla simple
- b) La comilla doble
- c) No se usan comillas
- d) Dos comillas simples

23. A los elementos de una estructura se les conoce como:

- a) Variables
- b) Miembros
- c) Constantes
- d) Índices

24. El caracter que separa la etiqueta de la estructura, con un elemento de la estructura es el:

- a) '.'
- b) '\*'
- c) '>'
- d) '<'



25. El tamaño de un arreglo depende entre otras cosas de:

- a) El tipo de dato
- b) Las variables usadas
- c) Las constantes usadas
- d) El compilador



## LO QUE APRENDÍ

Resuelve mediante un programa escrito en lenguaje C el siguiente problema: Una escuela nos solicita un programa para capturar las calificaciones de alumnos por materia, obtener el promedio por alumno, por materia y el promedio general del grupo. Emplea un arreglo multidimensional para almacenar las calificaciones de 6 materias obtenidas por 5 alumnos y realizar los cálculos solicitados.



# MESOGRAFÍA

## Bibliografía sugerida

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

## Bibliografía básica

Cairó, O. (2003). *Metodología de la Programación*. (2ª ed.) México: Alfaomega.

Ceballos, F.J. (1997). *Lenguaje C*. 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*. (3ª ed.) Madrid: Pearson / Prentice Hall.

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



## Bibliografía complementaria

López, L. (2004). *Programación estructurada, un enfoque algorítmico*. (2ª ed.) México: Alfa Omega.

Weiss, M.A. (2000) *Estructuras de datos en JAVA*. México: Addison Wesley.

## Sitios de Internet

Sitio	Descripción
<a href="http://www.lawebdelprogramador.com">www.lawebdelprogramador.com</a>	Documentos de temas de computación
<a href="http://www.monografias.com">www.monografias.com</a>	Documentos de temas de computación entre otros



SUAYED  
UNA OPCIÓN  
PARA TI

Licenciatura: Informática

## UNIDAD 6

# MANEJO DE APUNTADORES

# APUNTES DIGITALES PLAN 2012



SUAYED  
UNA OPCIÓN  
PARA TI



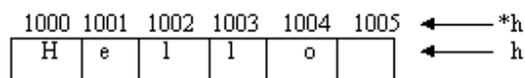
## OBJETIVO ESPECÍFICO

Al terminar la unidad, el alumno utilizará apuntadores en aplicaciones con arreglos, estructuras y funciones y podrá hacer uso dinámico de la memoria.



## INTRODUCCIÓN

Un 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.



SUAYED  
Sistema Universitario  
Autónomo de Uruguay

## LO QUE SÉ

Elabora una breve búsqueda acerca de los apuntadores, arreglos, estructuras, funciones y la memoria dinámica.



## TEMARIO DETALLADO

**(8 horas)**

- 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



## 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;**

### OPERADORES

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:

```
m= &cuenta;
```

```
q=*m;
```



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)
{
```

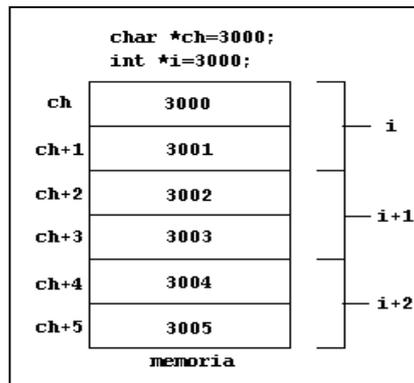


```
int x;  
int * p1, *p2;  
p1=&x;  
p2= p1;  
printf(“%p”,p2);  
}
```

### ARITMETICA

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.

```
p1++;  
p1--;  
p1+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.



<b>DECLARACIÓN</b>	tipo *nombre[n° elementos];
<b>ASIGNACIÓN</b>	nombre_arreglo[índice]=&variable;
<b>UTILIZAR ELEMENTOS</b>	*nombre_arreglo[índice];

### 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 n° 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[]={ "N° 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
```



SUAYED

// 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 (**->**).

### Ejemplos

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->télefono = 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.

La memoria se obtiene del *heap* (región de la memoria libre que queda entre el programa y la pila). El tamaño del *heap* es desconocido, pero contiene gran cantidad de memoria. El sistema de asignación dinámica está compuesto por las funciones *malloc* que asignan memoria a un puntero y *free* que libera la memoria asignada. Ambas funciones necesitan el archivo de cabecera *stdlib.h*.

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));
```



SUAYED

```
if(block_mem == NULL)
    printf("Memoria insuficiente\n");
else {
    printf("Memoria asignada\n");
    free(block_mem);
}

return(0);
}
```



## RESUMEN DE LA UNIDAD

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: el operador unario **&**, que devuelve la dirección de memoria de una variable, y el operador de referencia **\*** que devuelve el valor de la variable referenciada por el apuntador.

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.



## GLOSARIO DE LA UNIDAD

### **Apuntador**

Variable que contiene la dirección de memoria de otra variable.

### **Arreglo**

Conjunto de elementos del mismo tipo.

### **Estructura**

Conjunto de elementos de tipos distintos.

### **Función**

Subrutina que realiza tareas específicas.

### **Manejo dinámico de memoria**

Gestión de la memoria que permite asignar espacios memoria, y eliminar espacios memoria durante la ejecución de un programa.



## ACTIVIDADES DE APRENDIZAJE

### ACTIVIDAD 1

Realiza un programa que incremente en uno la posición de memoria a la que apunta.

### ACTIVIDAD 2

Realiza un programa que pase el valor de una variable entera a otra variable mediante el empleo de apuntadores.

### ACTIVIDAD 3

Elabora un programa que permita dar de alta y consultar los datos de un lote de autos usados, utiliza una estructura y apuntadores. Los datos a gestionar son: marca, año, kilometraje y precio.

### ACTIVIDAD 4

Elabora una función que sume dos números introducidos por el usuario, utiliza apuntadores.



### ACTIVIDAD 5

Elabora un programa que acepte el ingreso de cinco números, y que posteriormente imprima las direcciones de memoria de los valores correspondientes.



SUAYED  
SISTEMAS DE  
AYUDA AL  
ESTUDIANTE

## CUESTIONARIO DE REFORZAMIENTO

*Contesta el siguiente cuestionario.*

1. ¿Qué contiene un apuntador?
2. ¿Qué es la memoria principal?
3. ¿La memoria dinámica es sinónimo de *heap*?
4. ¿Cuál es el operador de dirección?
5. ¿Cuál es el operador de indirección?
6. ¿Cómo se declara un apuntador?
7. ¿Cuál es la ventaja de usar apuntadores con funciones?
8. ¿Cuál es la ventaja de usar apuntadores con arreglos?
9. ¿Cuál es la ventaja de usar apuntadores con estructuras?
10. ¿Qué es la aritmética de apuntadores?



## EXAMEN DE AUTOEVALUACIÓN

*I. En el espacio en blanco escribe la palabra que complete la oración.*

1. Un apuntador es una \_\_\_\_\_ que contiene una dirección de \_\_\_\_\_ de otra variable.
2. Los apuntadores siempre deben declararse, cuando éste no apunte a ninguna posición válida, ha de asignársele un valor \_\_\_\_\_.
3. El carácter \_\_\_\_\_ devuelve la dirección de memoria de su operando y \_\_\_\_\_ devuelve el contenido de la variable.
4. La suma y la resta son operadores \_\_\_\_\_ que se pueden usar como punteros.
5. Se puede utilizar la \_\_\_\_\_ para saber qué posición de memoria es más alta.



II. Lee con atención las siguientes preguntas y selecciona la respuesta correcta.

1. El \* devuelve:

- a) El contenido del operando
- b) La dirección de memoria de su operando
- c) Un tipo de dato
- d) Una variable

2. El & devuelve:

- a) El contenido del operando
- b) La dirección de memoria de su operando
- c) Un tipo de dato
- d) Una variable

3. Un arreglo que se declara como apuntador debe ir antecediendo por el caracter:

- a) &&
- b) ->
- c) \*
- d) &

4. Para leer un número en un arreglo que usa apuntadores, se usa:

- a) &
- b) %
- c) \*
- d) &&



5. Solo se necesita esta biblioteca para usar los apuntadores:

- a) string.h
- b) stdio.h
- c) conio.h
- d) stdlib.h

6. Para acceder a los miembros de una estructura se usa el:

- a) '&'
- b) '\*'
- c) '->'
- d) '<-'

7. Podemos definir a una estructura como un conjunto de:

- a) variables de tipos distintos
- b) variables
- c) variables del mismo tipo
- d) variables ordenadas

8. Podemos definir a una estructura como un conjunto de:

- a) variables de tipos distintos
- b) variables
- c) variables del mismo tipo
- d) variables ordenadas

9. La palabra reservada para definir una estructura es:

- a) define
- b) struct
- c) array
- d) char



10. Antes de definir una estructura se utiliza la palabra:

- a) define
- b) struct
- c) typedef
- d) int

11. Para desreferenciar a un apuntador se usa el:

- a) `'&'`
- b) `'*'`
- c) `'->'`
- d) `'<-'`

12. C pasa los parámetros de las funciones por:

- a) Valor
- b) Referencia
- c) Apuntadores
- d) Constantes

13. Si se utilizan apuntadores, los parámetros de las funciones se pasan

por:

- a) Valor
- b) Referencia
- c) Apuntadores
- d) Constantes



14. Si los parámetros de una función son apuntadores, se utiliza el signo de:
- a) '&'
  - b) '\*'
  - c) '->'
  - d) '<-'
15. El uso de apuntadores hace que un algoritmo de ordenación sea:
- a) Independientemente del tipo de dato
  - b) Más rápido
  - c) Más lento
  - d) Más eficiente
16. Un apuntador es:
- a) Una variable que contiene una dirección de memoria
  - b) Una dirección de memoria
  - c) Una variable
  - d) El signo de \*
17. Para determinar el tamaño en bytes de un tipo de dato se usa:
- a) free
  - b) sizeof
  - c) get
  - d) put
18. El \* se usa para:
- a) Acceder a una región en disco
  - b) Acceder a la dirección de memoria de una variable
  - c) Acceder al contenido de una variable



d) Acceder a una región en memoria

19. El & se utiliza para acceder a:

- a) una región en disco
- b) la dirección de memoria de una variable
- c) el contenido de una variable
- d) una región en memoria

20. Para asignar un espacio de memoria se usa la función:

- a) free
- b) malloc
- c) get
- d) put

21. Para liberar un espacio de memoria se usa:

- a) free
- b) malloc
- c) get
- d) typedef

22. Para incrementar una posición de memoria se puede usar:

- a) ->
- b) ++
- c) >
- d) \*



SUAYED

23. Para comparar dos apuntadores se usa el operador:

- a) &
- b) >
- c) \*
- d) \_>

24. Para inicializar un apuntador se puede usar:

- a) NULL
- b) &
- c) \*
- d) ->

25. El único entero que puede asignarse a un apuntador es:

- a) Sólo son tipos enteros
- b) Sólo son de tipo carácter
- c) Sólo son de tipo flotante
- d) Pueden no tener n tipo definido



## LO QUE APRENDÍ

Elabora un programa que, por medio de funciones controladas por un menú, permita agregar o quitar valores de una lista enlazada.



# MESOGRAFÍA

## Bibliografía sugerida

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

## Bibliografía básica

Ceballos, F.J. (1997). *Lenguaje C*. México: Alfaomega.

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

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



## Bibliografía complementaria

López, L. (2004). *Programación estructurada, un enfoque algorítmico*. (2ª. ed.) México: Alfa Omega.

Weiss, M. A. (2000) *Estructuras de datos en JAVA*. México: Addison Wesley.

## Sitios de Internet

Sitio	Descripción
<a href="http://www.cimat.mx/~alram/cpa/pointersC.pdf">http://www.cimat.mx/~alram/cpa/pointersC.pdf</a>	Tutorial sobre apuntadores y arreglos en C
<a href="http://expo.itch.edu.mx/view.php?f=c51">http://expo.itch.edu.mx/view.php?f=c51</a>	Apuntadores y estructuras
<a href="http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Manejo_din%C3%A1mico_de_memoria">http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Manejo_din%C3%A1mico_de_memoria</a>	Manejo dinámico de memoria
<a href="http://zarza.usal.es/~fgarcia/docencia/poo/02-03/P3.pdf">http://zarza.usal.es/~fgarcia/docencia/poo/02-03/P3.pdf</a>	Clases y manejo dinámico de memoria en C++



# RESPUESTAS A LOS EXÁMENES DE AUTOEVALUACIÓN

UNIDAD 1					
I		II		III	
1. V	6. F	11. F	1. c	1. d	6. a
2. V	7. V	12. V	2. e	2. a	7. c
3. F	8. V	13. F	3. d	3. a	8. d
4. V	9. F	14. F	4. b	4. c	9. a
5. F	10. V	15. F	5. f	5. c	
			6. a		

UNIDAD 2			
I	II		
1. a	1. d	9. d	17. c
2. f	2. a	10. a	18. d
3. g	3. c	11. d	19. a
4. b	4. c	12. d	20. c
5. h	5. a	13. c	21. d
6. d	6. a	14. c	
7. c	7. a	15. c	
8. e	8. c	16. c	



UNIDAD 3		
I	II	
1. V	1. a	6. a
2. F	2. a	7. a
3. V	3. c	8. b
4. V	4. b	9. a
5. F	5. c	10. b

UNIDAD 4			
I			
1. d	6. a	11. b	16. b
2. b	7. d	12. b	17. a
3. b	8. b	13. b	18. b
4. d	9. d	14. c	19. c
5. a	10. a	15. a	20. a

UNIDAD 5				
I				
1. a	6. c	11. c	16. b	21. d
2. a	7. c	12. b	17. a	22. b
3. b	8. a	13. c	18. a	23. b
4. a	9. b	14. c	19. a	24. a
5. b	10. a	15. b	20. d	25. a



SUAYED

UNIDAD 6					
I	II				
1. variable / memoria	1. a	6. c	11. b	16. a	21. a
2. nulo	2. b	7. a	12. a	17. b	22. b
3. & ó ampersand / * ó asterisco	3. c	8. a	13. b	18. c	23. b
4. aritméticos	4.a	9. b	14. b	19. b	24. a
5. comparación	5. b	10.c	15. a	20. b	25. d