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 2011

SUAyED UNA OPCIÓN PARA TI

INTRODUCCIÓN A LA PROGRAMACIÓN

Clave:		Créditos: 8
Licenciatura: INFORMÁTICA		Semestre: 2º
Área: Informática (Desarrollo de sistemas)		Horas. Asesoría:
Requisitos: Ninguno		Horas. por semana: 4
Tipo de asignatura:	Obligatoria (x)	Optativa ( )

AUTOR:

DAVID ESPARTACO KANAGUSICO HERNÁNDEZ

ADAPTADO A DISTANCIA:

ACTUALIZACION AL PLAN DE ESTUDIOS 2011:

DAVID E. KANAGUSICO HDZ.

TEMARIO OFICIAL
(64 horas)

	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

INTRODUCCIÓN GENERAL A LA ASIGNATURA

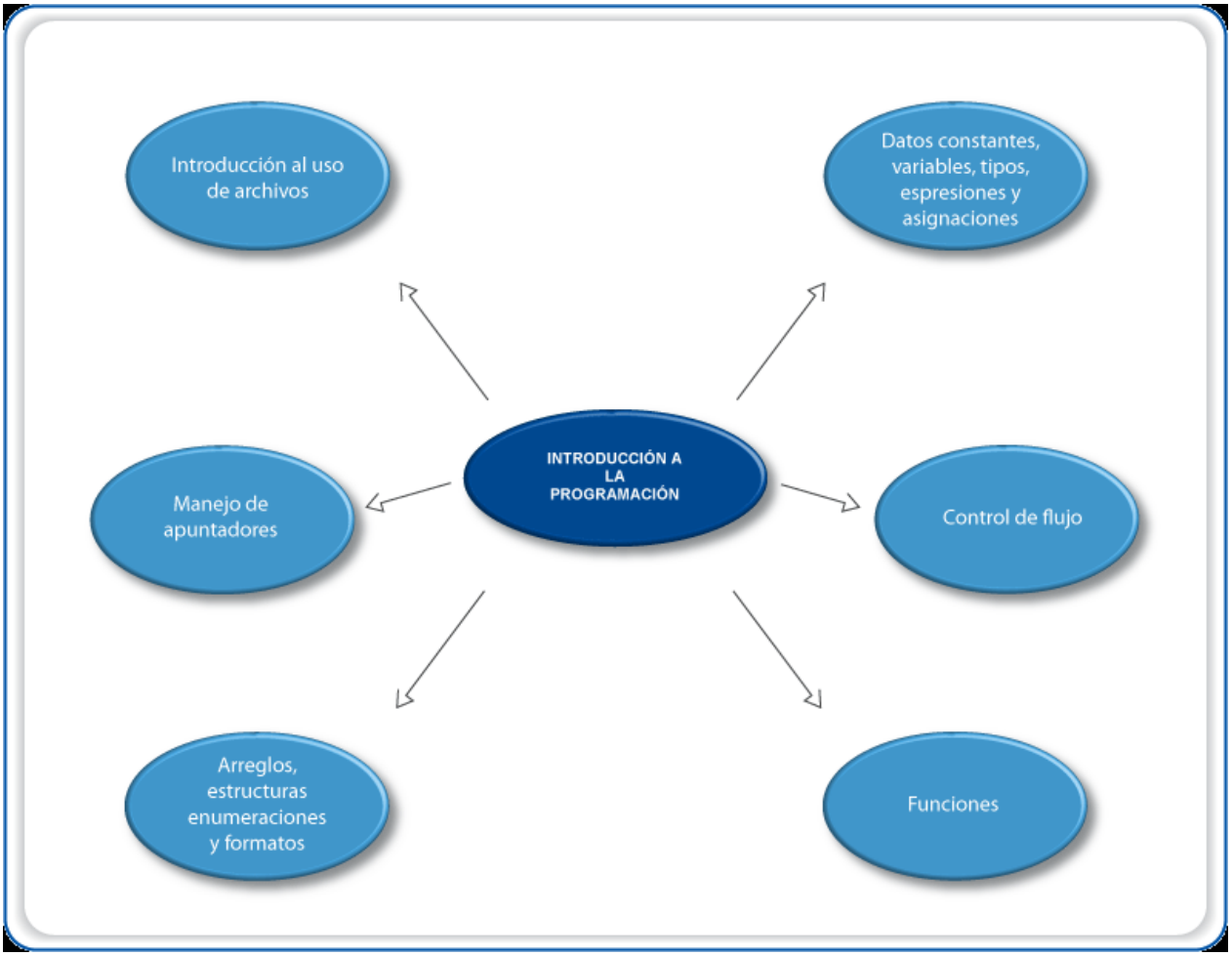
Las notas explican los puntos necesarios para el desarrollo de programas de computadora. Se tratan conceptos básicos y de la estructura de un programa, además de temas avanzados como son el uso de apuntadores y archivos.

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 DE LA ASIGNATURA

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

ESTRUCTURA CONCEPTUAL



UNIDAD 1

INTRODUCCIÓN A LA PROGRAMACIÓN

OBJETIVO ESPECÍFICO

Al terminar la unidad, el alumno será capaz establecer la diferencia entre los paradigmas de programación e identificar los lenguajes de acuerdo a 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 si 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.

Un lenguaje de programación, es una herramienta que permite desarrollar programas para computadora. Dichos lenguajes permiten expresar las instrucciones que el programador desea ejecutar.

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

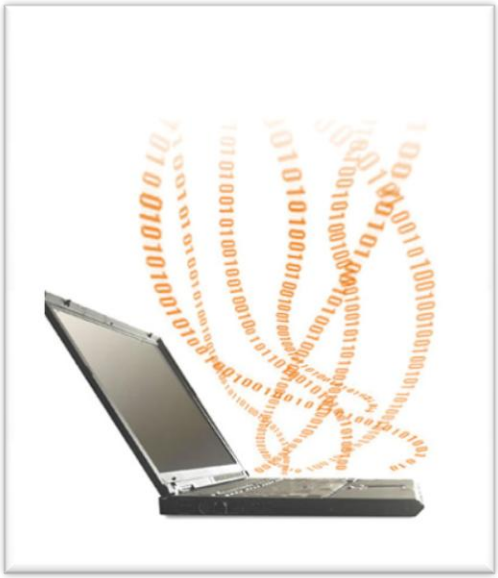
Existen, además, dos herramientas muy utilizadas para transformar las instrucciones de un lenguaje de programación a código máquina. Estas herramientas se denominan intérpretes y compiladores.

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.

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.

Envía tus respuestas al asesor.

TEMARIO DETALLADO
(4 horas)

1.1 Concepto de lenguaje de programación

1.2 Paradigmas de programación

- Paradigma imperativo
- Paradigma orientado a objetos
- 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).

Nota: ***Para efectos del curso se utilizará lenguaje C***.

En particular, este lenguaje está caracterizado por ser de uso general, de sintaxis compacta y portable.

Características del lenguaje C

El lenguaje C es **de uso general**, ya que puede ser usado 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, 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

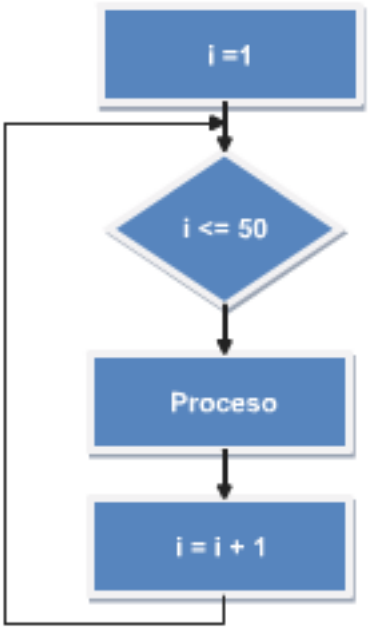
→ Paradigma imperativo

La programación imperativa es una forma de escribir programas secuenciales; significa escribir un programa en base a 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. **C** también es un lenguaje estructurado. Cabe destacar que en el lenguaje C han sido desarrollados la mayoría de los sistemas operativos, manejadores de bases de datos y aplicaciones de la actualidad.

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
Declaraciones globales
tipo_devuelto main (parámetros)
{
sentencias(s);
}

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 un mensaje en pantalla. La primera línea que aparece se denomina *Encabezado* y es un archivo que proporciona información al compilador. En este archivo están incluidas las funciones *printf.*

La segunda línea del programa indica dónde comienza la función *main* indicando los valores que recibe y los que devuelve.

printf muestra un mensaje en la pantalla.

Por último, la sentencia *return*(0) indica que esta función no devuelve ningún 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);
}
```

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

→ 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 a la influencia de C++ (una extensión del lenguaje C). Su dominación fue consolidada gracias al auge de las interfases 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, 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
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
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
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
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.

→ Paradigma funcional

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

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

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

Cuando se aplica una función, los argumentos que esta 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. Una definición de función especifica el dominio y rango, de manera implícita o explicita, 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.

1.3 Lenguaje máquina

Los circuitos micro programables son sistema 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 dichos signos.

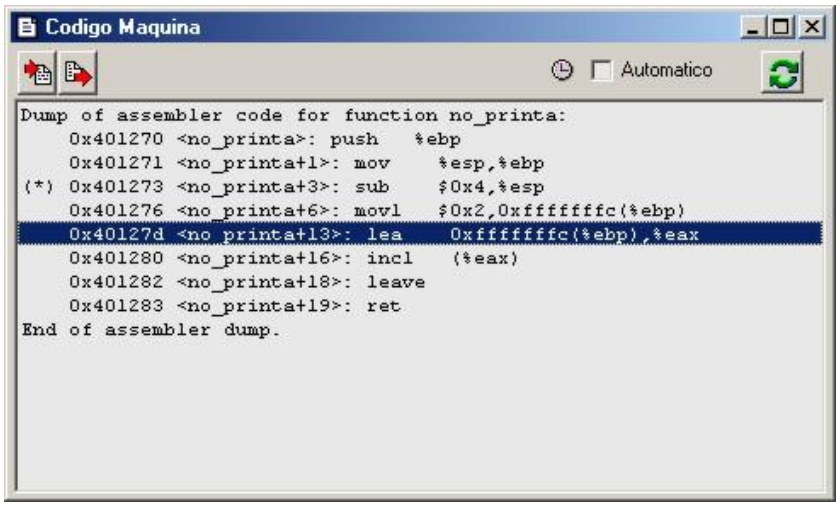
El lenguaje de máquina es el sistema de códigos directamente interpretable por un circuito microprogramable, como el microprocesador de una computadora. Este lenguaje está compuesto por un conjunto de instrucciones que determinan acciones que serán por la máquina. Un programa de computadora consiste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

1.4 Lenguajes de bajo nivel

Por otro lado, 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 trasladado al lenguaje máquina.

El término ensamblador (del inglés *assembler*) se refiere a un tipo de programa informático que se encarga de traducir un archivo fuente escrito en un lenguaje ensamblador, a un archivo objeto que contiene código máquina, ejecutable directamente por la máquina para la que se ha generado.



1.5 Lenguajes de alto nivel

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

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

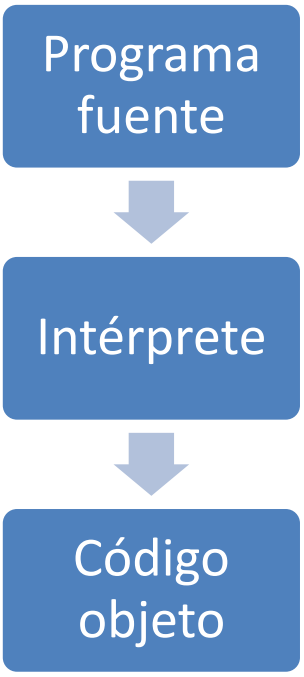
Ejemplos de lenguajes de alto nivel	
* C++.	Es un lenguaje de programación, diseñado a mediados de los años 80, por Bjarne Stroustrup. Por otro lado, es un lenguaje que abarca dos paradigmas de la programación: la programación estructurada y la programación orientada a objetos.
* Fortran	Es un lenguaje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

	para sitios web. Últimamente se usa también para la creación de otro tipo de programas incluyendo aplicaciones con interfaz gráfica usando las librerías Qt o GTK+.
* Python	Es un lenguaje de programación creado por Guido van Rossum en el año 1990. En la actualidad Python se desarrolla como un proyecto de código abierto, administrado por la Python Software Foundation. La última versión estable del lenguaje es actualmente (septiembre 2006) la 2.5.

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.

Como ejemplo de lenguajes interpretados tenemos a: PHP, Perl y Python



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.

Ejemplo de lenguajes que utiliza 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.

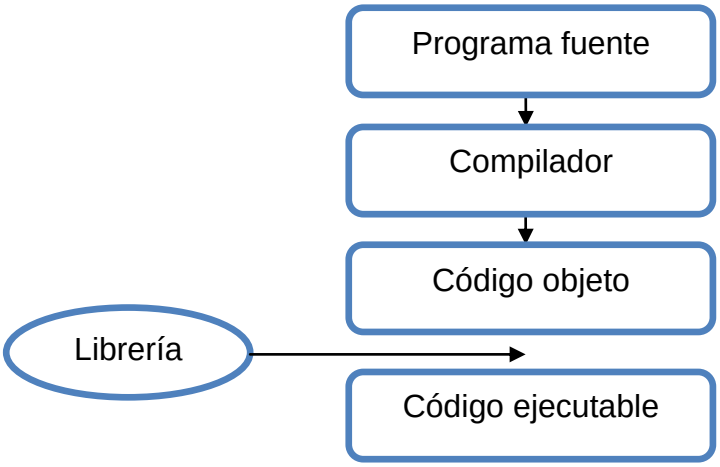
1.8 Fases de la compilación

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 de la foma:

<simbolo> :: = <expresión con simbolos>

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

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

Llos 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 inidcan el órden 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.*
- Simbolos 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 desarrollan los conceptos básicos de la programación, entendida ésta como la implementación de un algoritmo (serie de pasos para resolver un problema) en un lenguaje de programación, dando como resultado un programa. Se trataron diversos temas relacionados con la programación, como la programación estructurada, además del funcionamiento de intérpretes y compiladores.

GLOSARIO

Algoritmo

Conjunto de pasos para resolver un problema.

Clase

Una clase es un conjunto de objetos que comparten características comunes.

Compilador

Programa 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 mas 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).

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.

Paradigma

Un paradigma es un modelo o patrón a seguir.

Parámetros de una función

Conjunto de variables que utiliza una función.

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.

ACTIVIDADES DE APRENDIZAJE

#	Actividad
1.	Realiza una investigación de la historia del lenguaje de programación C.
2.	Realiza una investigación acerca de las cuatro primeras generaciones de los lenguajes de programación.
3.	Realiza un algoritmo que sume tres números. Si necesitas mayor información de cómo realizar un algoritmo visita el sitio http://www.lawebdelprogramador.com/
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);
}
```

5.	Investiga los conceptos de objeto, clase y herencia. Apóyate en el documento El paradigma orientado a objetos **(ANEXO 2).** No olvides colocar tus fuentes bibliográficas.
6.	Realiza un cuadro comparativo de la programación orientada a objetos y la programación imperativa.
7.	Realiza una investigación sobre: • Lenguaje ensamblador • Lenguaje de bajo nivel • Lenguaje de alto nivel Identifica sus iniciadores, conceptos, principales características, diferencias entre estos lenguajes, ventajas/desventajas, estructura, y proporciona 2 ejemplos de cada uno.
8.	Realiza un programa en lenguaje ensamblador que sume dos números.
9.	Investiga y enlista 5 lenguajes de programación que sean de alto nivel.
10.	Investiga 5 lenguajes de programación que utilicen un intérprete y 5 que empleen compiladores.
11.	Investiga y elabora un cuadro comparativo de las diferencias entre el intérprete y el compilador.
12.	Realiza un programa en C e identifica los pasos que realiza el compilador para generar el programa ejecutable.

CUESTIONARIO DE REFORZAMIENTO

Contesta el siguiente cuestionario.

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?
6. ¿Qué es un algoritmo?
7. ¿Qué es un programa?
8. ¿Qué es un compilador?
9. ¿Qué es un interpreté?
10. ¿Qué es un objeto?

Realiza el cuestionario en un procesador de textos y envíalo a tu asesor.

LO QUE APRENDÍ

Realiza un programa en C que transforme una cantidad en grados Fahrenheit a su equivalente en grados centígrados.

Realiza tu actividad en un procesador de textos y envíalo a tu asesor.

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. Printf muestra un mensaje en pantalla.	( )	( )
5. Scanf muestra un mensaje en pantalla.	( )	( )

II. Indica si las oraciones son verdaderas o falsas.

	Verdadera	Falsa
1. La programación estructurada utiliza objetos	( )	( )
2. C es un lenguaje orientado a objetos.	( )	( )
3. Un ejemplo de una estructura de control es la iteración	( )	( )
4. Una función es sinónimo de una clase.	( )	( )
5. Scanf muestra un mensaje en pantalla.	( )	( )

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

☐ 1. Este lenguaje se refiere a un sistema de códigos directamente interpretable por un circuito micro programable. ☐ 2. Consiste en una cadena de instrucciones de lenguaje de máquina. ☐ 3. Este lenguaje proporciona poca o ninguna abstracción del microprocesador de una computadora. ☐ 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. ☐ 5. Se caracteriza por expresar los algoritmos de una manera adecuada a la capacidad cognitiva humana. ☐ 6. Es ejemplo de un lenguaje de alto nivel.	A PHP B Ensamblador C Lenguaje de maquina D Lenguaje de bajo nivel E Programa de computadoras F Lenguaje de alto nivel

IV. Indica si las oraciones son verdaderas o falsas.

	Verdadera	Falsa
1. Todos los leguajes de programación usan compatibilidades	( )	( )
2. C ++ es un lenguaje orientado a objetos.	( )	( )
3. La programación orientada a objetos divide un problema complejo en problemas más sencillos	( )	( )
4. El lenguaje PHP utiliza un compilador.	( )	( )
5. El lenguaje C es un ejemplo de un lenguaje de bajo nivel.	( )	( )

V. Selecciona la opción que consideres correcta:

1. El código fuente es:

- ❍ a) Un código de computadora.
- ❍ b) Un diagrama de flujo.
- ❍ c) Un paradigma.
- ❍ d) Un conjunto de caracteres entendibles por un ser humano.

2. El código objeto es:

- ❍ a) Un código entendible por la computadora.
- ❍ b) Sinónimo de un interpreté.
- ❍ 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 interpreté.
- ❍ 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) Conjunto de funciones con características similares.
- ❍ d) 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 interpreté:

- ❍ a) Lee línea por línea el código fuente
- ❍ b) Lee línea por línea el código objeto.
- ❍ c) Lee línea por línea el código ejecutable.
- ❍ d) Lee línea por línea las librerías.

7. Ejemplo de una librería:

- ❍ a) scanf
- ❍ b) void
- ❍ c) stdio.h
- ❍ d) Java.

8. Python utiliza un:

- ❍ a) Un enlazador.
- ❍ b) Un compilador y un interpreté.
- ❍ c) Compilador.
- ❍ d) Interpreté.

9. Un linker:

- ❍ a) Agrega librerías.
- ❍ b) Genera un código objeto.
- ❍ c) Genera un código fuente.
- ❍ d) Es sinónimo de clase

MESOGRAFÍA

BIBLIOGRAFÍA BÁSICA

1. Cairo, Osvaldo, *Metodología de la Programación*, 2a ed., México, Alfaomega, 2003, 438 pp.

2. Ceballos, Francisco Javier. *Lenguaje C*, Alfaomega, 1997, 884 pp.

3. Deitel, H.M., Deitel, P.J. *Como programar en C/C++ y Java*, 4a ed., México: Prentice Hall, 2004, 1113 pp.

4. Joyanes, Luis. *Fundamentos de Programación*, 3a Edición, España: Pearson Prentice Hall, 2003, 1004 pp.

BIBLIOGRAFÍA COMPLEMENTARIA

1. García, Luis, Juan Cuadrado, Antonio De Amescua y Manuel Velasco, Construcción lógica de programas, Teoría y problemas resueltos, México, coedición Alfa omega-RaMa, 2004, 316 pp.

2. López, Leobardo, Programación estructurada en turbo pascal 7, México, Alfa omega, 2004, 912 pp.

3. López, Leobardo, Programación estructurada, un enfoque algorítmico, 2ª. Ed., México, Alfa omega, 2004, 664 pp.

4. Sedgewick, Robert, Algoritmos en C++, México, Adisson-Wesley Iberoamericana, 1995, 800 pp.

SITIOS ELECTRÓNICOS

Sitio	
– www.monografias.com	– www.wikipedia.org
– www.lawebdelprogramador.com	

UNIDAD 2

TIPOS DE DATOS ELEMENTALES

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

OBJETIVO ESPECÍFICO

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

INTRODUCCIÓN

Un tipo de dato lo podemos definir a partir de sus valores permitidos (entero, carácter, etc.), por otro lado, las palabras reservadas son utilizadas por 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.

Envía tus respuestas a tu asesor.

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

Un tipo de dato lo podemos definir a partir de sus valores permitidos (entero, carácter, etc.); por otro lado, las palabras reservadas son utilizadas por un lenguaje de programación para fines específicos, y no pueden ser utilizados 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.

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 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 32.767	2	Para números enteros.
unsigned int	0 a 65.535	2	Para números enteros.
long int	±2.147.483.647	4	Para números enteros
unsigned long int	0 a 4.294.967.295	4	Para números enteros
float	3.4E-38 decimales(6)	6	Para números con decimales
double	1.7E-308 decimales(10)	8	Para números con decimales
long double	3.4E-4932 decimales(10)	10	Para números con decimales

Ejemplo de utilización de tipos de datos en C.

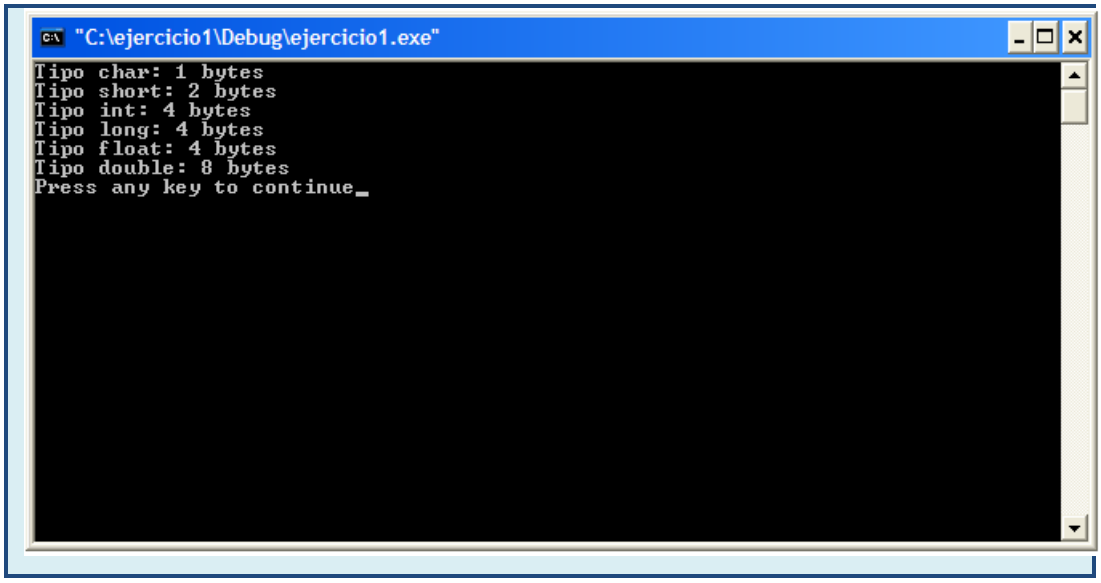
```
int numero;

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:


"C:\ejercicio1\Debug\ejercicio1.exe"
Tipo char: 1 bytes
Tipo short: 2 bytes
Tipo int: 4 bytes
Tipo long: 4 bytes
Tipo float: 4 bytes
Tipo double: 8 bytes
Press any key to continue

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

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.

2.3 Identificadores

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 caractreres, 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 valido 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 aritmeticos

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
b) Relacionales
c) Lógicas

→ **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 dentro hacia afuera, 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) = 28.98	2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98

2.6 Variables y constantes

Constantes

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

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

Hay constantes de varios tipos:

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.

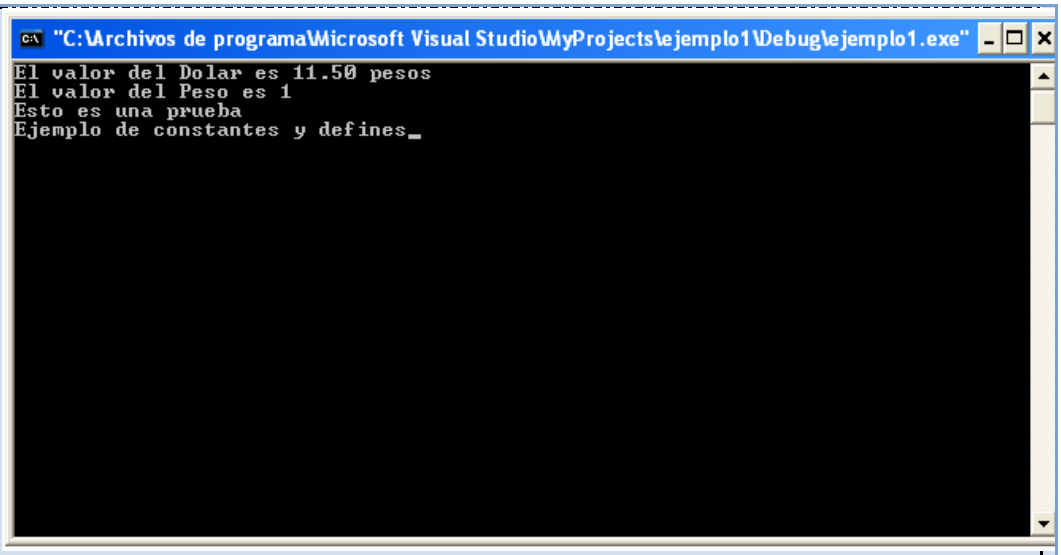
SINTAXIS:

```
#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();
}
```

El resultado del programa es el siguiente:



Variables

Unidad básica de almacenamiento. 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;
```

Conversion

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 a convertir mantiene su valor.

SINTAXIS:

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

Ejemplo:

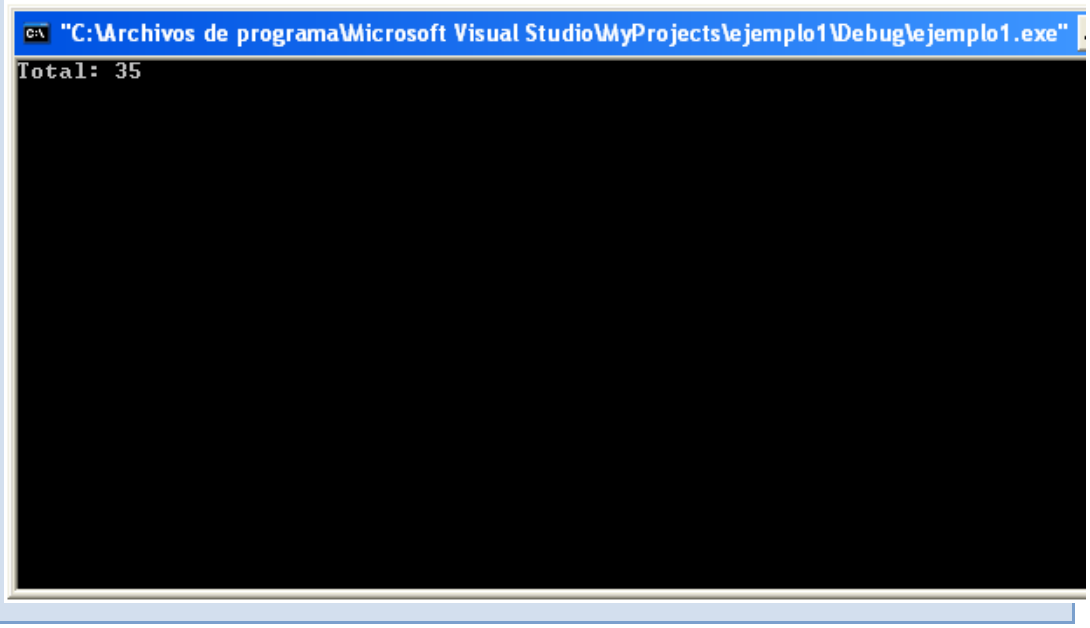
Convertimos 2 variables *float* para guardar la suma en un entero.

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:



Ambito 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:

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 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 verán los conceptos de variable, constante, expresión y palabra, así como ejemplos de su utilización en lenguajes de programación, tomando como ejemplo el lenguaje C.

GLOSARIO

Constante

Un valor que no cambia durante la ejecución del programa.

Dato

Elemento más pequeño con un significado propio.

Expresión

Conjunto de variables, constantes 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.

Variable

Un valor que cambia constantemente 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.

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++
2.	Investiga los tipos de datos del lenguaje java.
3	Investiga las siguientes funciones: Función de la palabra reservada *enum* Función de los identificadores
4	Investiga las palabras reservadas del lenguaje PHP.
5	Realiza un programa en C que determine si un número es par o impar.
6	Realiza un programa en C que determine si dos números introducidos por el usuario son iguales.
7	Investiga cuál es la prioridad del operador de exponente en el lenguaje Visual Basic.
8	Investiga el uso de variables locales y globales en Python.
9	Investiga el uso de constantes en Java.
10	Investiga y escribe la estructura de un programa escrito en Visual Basic.

CUESTIONARIO
DE REFORZAMIENTO

Responde 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. ¿Qué significa la palabra switch?
10. ¿Qué es una conversión de tipos?

Realiza el cuestionario en un procesador de textos y envíalo a tu asesor.

LO QUE APRENDÍ

Realiza dos programas en C de acuerdo a lo siguiente:

1. Que transforme una cantidad introducida en pesos, a su equivalente en dólares y euros.

2. Que calcule el área de un triangulo, utiliza variables y constantes.

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

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 corresponde.

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

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

II. Elige de entre las opciones, la respuesta que consideres correcta:

1. Un tipo de dato:

- a) Permite usar varios elementos en una estructura
- b) Determina las clases a usar
- c) Determina los objetos a 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 digito
- c) Números con decimales
- d) Números y letras

4. La siguiente es una palabra reservada:

- a) Si
- b) Entonces
- c) If
- d) Mientras

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 esta 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 modulo 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) !

19. Es ejemplo de una biblioteca:

- a) stdio.h
- b) nclude
- 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;	
>=	+
%	

MESOGRAFÍA

BIBLIOGRAFÍA BÁSICA

1. Cairo, Osvaldo. ”Metodología de la Programación”, México, Alfaomega, 2a Edición, 2003, 438 pp.

2. Ceballos, Francisco Javier. ”Lenguaje C”, Alfaomega, 1997, 884 pp.

3. Deitel, H.M., Deitel, P.J. “Como programar en C/C++ y Java”, México: Prentice Hall, 4a Edición, 2004, 1113 pp.

4. Joyanes, Luis. “Fundamentos de Programación”, España: Pearson Prentice Hall, 3a Edición, 2003, 1004 pp.

BIBLIOGRAFÍA COMPLEMENTARIA

1. García, Luis, Juan Cuadrado, Antonio De Amescua y Manuel Velasco, Construcción lógica de programas, Teoría y problemas resueltos, México, coedición Alfa omega-RaMa, 2004, 316 pp.
2. López, Leobardo, Programación estructurada en turbo pascal 7, México, Alfa omega, 2004, 912 pp.
3. López, Leobardo, Programación estructurada, un enfoque algorítmico, 2ª. Ed., México, Alfa omega, 2004, 664 pp.
4. Sedgewick, Robert, Algoritmos en C++, México, Adisson-Wesley Iberoamericana, 1995, 800 pp.
5. Weiss, Mark Allen, Estructuras de datos en JAVA, México, Addison Wesley, 2000, 740 pp.

SITIOS ELECTRÓNICOS

Sitio	Descripción
www.lawebdelprogramador.com	Documentos de temas de computación
www.monografias.com	Documentos de temas de computación entre otros

UNIDAD 3

CONTROL DE FLUJO

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 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
- Condicional
- Iterativa.

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. Coméntalo con tu asesor.

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

Inicio
Acción a_1
Acción a_2
.
.
Acción a_n
Fin

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

Ejercicio.

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 el anexo **(ANEXO 4).**

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 (expresion-booleana)
        Si condición Entonces   {
Acción                                  Sentencia1;
De_lo_contrario                         sentencia2;
Acción                          }
Fin_Si                          else
Fin                                     Sentencia3;
```

IF-ELSE

La ejecución atraviesa un conjunto de estados booleanos que determinan que se ejecuten distintos fragmentos de código.

```
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**.

Ejercicio

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

```
# include <stdio.h>
main()
{
  float examen, tareas, trabajo, final;
  printf("Por favor introduzca la calificación de los exámenes: ");
  scanf("%f",&examen);
  printf("Por favor introduzca la calificación de las tareas: ");
  scanf("%f",&tareas);
  printf("Por favor introduzca la calificación del trabajo: ");
  scanf("%f",&trabajo);
  final = (examen+tareas+trabajo)/3;
  printf("Tu calificación final es de: %.2f\n",final);
  if(final < 6)
    printf("Tendrás que cursar programación nuevamente\n");
  else
    printf("Aprobaste con la siguiente calificación: %.2f\n",final);
  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 ésta es mucho más cómoda y fácil de comprender. Si los valores con los que se compara son números se pone directamente, pero si es un carácter se debe encerrar entre comillas simples.

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

```
if (expresion-booleana)
{
    Sentencia1;
    sentencia2;
}
Else
Sentencia3;
```

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

La sentencia *break* realiza la salida de un bloque de código. En el caso de sentencia switch, realiza el código y cuando ejecuta *break*, sale de este bloque y sigue con la ejecución del programa. En el caso 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, en caso de que el usuario introduzca un número incorrecto aparecerá el mensaje “Opción incorrecta”.

Ejercicio

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
  int opcion;
  printf("1.ALTAS\n");
  printf("2.BAJAS\n");
  printf("3.MODIFICA\n");
  printf("4.SALIR\n");
  printf("Elegir opción: ");
  scanf("%d",&opcion);
  switch(opcion)
  {
    case 1:
        printf("Opción Uno");
    break;
    case 2:
        printf("Opción Dos");
    break;
    case 3:
        printf("Opción Tres");
    break;
    case 4:
        printf("Opción Salir");
    break;
```

```
            default:
        printf("Opción incorrecta");
      }
    getch();
    }
```

Observa más **ejemplos de estructura alternativa** en el anexo 5 **(ANEXO 5).**

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.

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

```
while(condición){

cuerpo;

}
```

Ejercicio

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);
```

Ejercicio

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

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

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

Para ver más ejemplos de estos ejercicios, remitirse al **anexo 6** **(ANEXO 6)**.

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;fin;iteración)

    sentencia1;
```

Ejercicio

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();
}
```

Para ver más **ejemplos** de estos **ejercicios**, remitirse al anexo **(ANEXO 7).**

RESUMEN DE LA UNIDAD

En esta unidad se verán las características de la estructura secuencial, la estructura repetitiva y la estructura alternativa, además se verán ejemplos de su utilización en el lenguaje C.

GLOSARIO

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.	Realiza un programa en C que obtenga la edad de una persona utilizando como base, la fecha de nacimiento.
2.	Realiza 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.
3	Realiza un programa en lenguaje C que sume dos números, con la condición de que dichos números sean mayores a cero.
4	Realiza un programa que determine si una persona es menor de edad, adulto, o adulto mayor, de acuerdo a un número introducido por el usuario.
5	Realiza un programa que sume dos números mientras dichos números sean mayores a cero.
6	Realiza un programa que obtenga la potencia de un número, mientras dicho número sea mayor a cero.

CUESTIONARIO
DE REFORZAMIENTO

Contesta el siguiente cuestionario.

1. Define qué es una estructura secuencial.
2. Define qué es una estructura alterativa.
3. Define a 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 palabra *default*?
8. ¿Cuál es la función de la palabra *switch*?
9. ¿Cuál es la función de la palabra *case*?
10. ¿Cuál es la función de la palabra *break*?

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

LO QUE APRENDÍ

Realiza un programa que, a través de un menú, obtenga el área de un triangulo, el área de un rectángulo y el área de un circulo, mientras los valores introducidos por el usuario sean mayores a cero.

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

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 while.	( )	( )
3. Si la condición no se cumple se puede utilizar la palabra else para cambiar el flujo.	( )	( )
4. Una opción para varios if es utilizar la palabra reservada case.	( )	( )
5. Dentro del case 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:

- ❍ a) Un ciclo con un número determinado de iteraciones
- ❍ b) Un ciclo con un número indeterminado de iteraciones
- ❍ c) Un ciclo infinito
- ❍ d) Un ciclo 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

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

MESOGRAFÍA

BIBLIOGRAFÍA BÁSICA

1. Cairo, Osvaldo. ”Metodología de la Programación”, México, Alfaomega, 2a Edición, 2003, 438 pp.

2. Ceballos, Francisco Javier. ”Lenguaje C”, Alfaomega, 1997, 884 pp.

3. Deitel, H.M., Deitel, P.J. “Como programar en C/C++ y Java”, México: Prentice Hall, 4a Edición, 2004, 1113 pp.

4. Joyanes, Luis. “Fundamentos de Programación”, España: Pearson Prentice Hall, 3a Edición, 2003, 1004 pp.

BIBLIOGRAFÍA COMPLEMENTARIA

1. López, Leobardo, Programación estructurada, un enfoque algorítmico, 2ª. Ed., México, Alfa omega, 2004, 664 pp.

2. Sedgewick, Robert, Algoritmos en C++, México, Adisson-Wesley Iberoamericana, 1995, 800 pp.

3. Weiss, Mark Allen, Estructuras de datos en JAVA, México, Addison Wesley, 2000, 740 pp.

SITIOS ELECTRÓNICOS

Sitio
www.lawebdelprogramador.com
www.monografias.com

UNIDAD 4

FUNCIONES

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 e**s 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.

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

Las funciones son los bloques constructores de C, 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.

Es la característica más importante de C. 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

En base a lo que hemos visto en las unidades anteriores y a 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 preguúntala a tu asesor, y una vez concluido, envíaselo por correo.

TEMARIO DETALLADO
(18 horas)

4.1 Internas

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.

→ 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.

Veamos las funciones para el manejo de caracteres.

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

Veamos un ejercicio donde se utiliza la función isalpha.

Ejercicio

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 *in* 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.

memset	Inicializar una región de memoria (buffer) con un valor determinado. Se utiliza principalmente para inicializar cadenas con un valor determinado. Cabecera <string.h>.	memset (var_cadena,'carácter',tamaño);
strcat	Concatena cadenas, es decir, añade la segunda a la primera, que no pierde su valor original. Hay que tener en cuenta que la longitud de la primera cadena debe ser suficiente para guardar la suma de las dos cadenas. Cabecera <string.h>.	strcat(cadena1,cadena2);
strchr	Devuelve un puntero a la primera ocurrencia del carácter especificado en la cadena donde se busca. Si no lo encuentra, devuelve un puntero nulo. Cabecera <string.h>.	strchr(cadena,'carácter'); strchr("texto",'carácter');

strcmp	Compara alfabéticamente dos cadenas y devuelve un entero basado en el resultado de la comparación. La comparación no se basa en la longitud de las cadenas. Muy utilizado para comprobar contraseñas. Cabecera <string.h>.	strcmp(cadena1,cadena2);
		strcmp(cadena2,"texto");
		RESULTADO
		/ALOR — DESCRIPCIÓN
		Menor a Cero — Cadena1 menor a Cadena2.
		Cero — Las cadenas son iguales.
		Mayor a Cero — Cadena1 mayor a Cadena2.

strcpy:	Copia el contenido de la segunda cadena en la primera. El contenido de la primera se pierde. Lo único que debemos contemplar es que el tamaño	strcpy(cadena1,cadena2); strcpy(cadena1,"texto");

	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>	variable=strlen(cadena);
strncat	Concatena el número de caracteres de la segunda cadena en la primera. Ésta última no pierde la información. Hay que controlar que la longitud de la primera cadena tenga longitud suficiente para guardar las dos cadenas. Cabecera <string.h>.	strncat(cadena1,cadena2,nº de caracteres);
strncmp	Compara alfabéticamente un número de caracteres entre dos cadenas. Devuelve un entero según el resultado de la comparación. Los valores devueltos son los mismos que en la función strcmp. Cabecera <string.h>.	strncmp(cadena1,cadena2,nº de caracteres); strncmp(cadena1,"texto",nº de caracteres);
strncpy	Copia un número de caracteres de la segunda cadena a la primera. En la primera cadena se pierden aquellos caracteres que se copian de la segunda. Cabecera <string.h>.	strncpy(cadena1,cadena2,nº de caracteres); strncpy(cadena1,"texto",nº de caracteres);
strrchr	Devuelve un puntero a la última ocurrencia del carácter buscado en la cadena. Si no lo encuentra devuelve un puntero nulo. Cabecera <string.h>.	strrchr(cadena,'carácter'); strrchr("texto",'carácter');
strpbrk	Devuelve un puntero al primer carácter de la cadena que coincida con otro de la cadena a buscar. Si no hay correspondencia devuelve un puntero nulo. Cabecera <string.h>.	strpbrk("texto","cadena_de_busqueda"); strpbrk(cadena,cadena_de_busqueda);
tolower	Devuelve el carácter equivalente al de la variable en minúsculas si la variable es una letra; y no la cambia si la letra es minúscula. Cabecera <ctype.h>.	variable_char=tolower(variable_char);

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

Para ver ejemplos de ejercicios de cadenas, revisa el anexo que contiene **Ejercicios de Cadenas (ANEXO 8).**

→ 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 *double*. Muestra el arcocoseno de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>.	double acos(variable_double);
asin	Devuelve un tipo *double*. Muestra el arcoseno de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>.	double asin(variable_double);
atan	Devuelve un tipo *double*. Muestra el arcotangente de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera <math.h>.	double atan(variable_double);
cos	Devuelve un tipo *double*. Muestra el coseno de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera <math.h>.	double cos(variable_double_radianes);
sin	Devuelve un tipo *double*. Muestra el seno de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera <math.h>.	Doublé sin(variable_double_radianes);
tan	Devuelve un tipo *double*. Muestra la tangente de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera <math.h>.	double tan(variable_double_radianes);
cosh	Devuelve un tipo *double*. Muestra el coseno hiperbólico de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.	double cosh(variable_double);
sinh	Devuelve un tipo *double*. Muestra el seno hiperbólico de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.	double sinh(variable_double);
tanh	Devuelve un tipo *double*. Muestra la tangente hiperbólica de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser <math.h>.	double tanh(variable_double);
ceil	Devuelve un *double* que representa el menor	double ceil(variable_double);

	entero sin serlo más que la variable redondeada. Por ejemplo, dado 1.02 devuelve 2.0. Si asignamos –1.02, devuelve –1. En resumen, redondea la variable a la alta. Cabecera <math.h>.	
floor	Devuelve un *double* que representa el entero mayor, sin serlo más quela variable redondeada. Por ejemplo dado 1.02 devuelve 1.0. Si asignamos –1.2 devuelve –2. En resumen redondea la variable a la baja. Cabecera <math.h>.	double floor(variable_double);
fabs	Devuelve un valor *float* o *double*. Devuelve el valor absoluto de una variable *float*. Se considera una función matemática, pero su cabecera es <stdlib.h>.	var_float fabs(variable_float);
labs	Devuelve un valor *long*. Devuelve el valor absoluto de una variable long. Se considera una función matemática, pero su cabecera es <stdlib.h>.	var_long labs(variable_long);
abs	Devuelve un valor entero. Devuelve el valor absoluto de una variable int. Se considera una función matemática, pero su cabecera es <stdlib.h>.	var_float abs(variable_float);
modf	Devuelve un *double*. Descompone la variable en su parte entera y fraccionaria. La parte decimal es el valor que devuelve la función, su parte entera la guarda en el segundo término de la función. Las variables tienen que ser obligatoriamente de tipo *double*. Cabecera <math.h>.	var_double_decimal= modf(variable,var_parte_entera);
pow	Devuelve un *double*. Realiza la potencia de un número base elevada a un exponente que nosotros le indicamos. Se produce un error si la base es cero o el exponente es menor o igual a cero. La cabecera es <math.h>.	var_double=pow(base_double,exponente_double);
sqrt	Devuelve un *double*. Realiza la raíz cuadrada de la variable, ésta no puede ser negativa, si lo es se produce un error de dominio. La cabecera <math.h>.	var_double=sqrt(variable_double);

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

Ejercicio

Este programa imprime las diez primeras potencias de 10

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

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

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

  return 0;
}
```

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

→ Funciones de conversión

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

atoi	Devuelve un entero. Esta función convierte la cadena en un valor entero. La cadena debe contener un número entero válido, si no es así el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es <stdlib.h>.	int atoi(variable_char);
atol	Devuelve un *long*. Esta función convierte la cadena en un valor *long*. La cadena debe contener un número long válido, si no es así, el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es **<stdlib.h>**.	long atol(variable_char);
atof	Devuelve un *double*. Esta función convierte la cadena en un valor *double*. La cadena debe contener un número *double* válido, si no es así, el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es **<stdlib.h>**.	double atof(variable_char);
sprintf	Devuelve una cadena. Esta función convierte cualquier tipo numérico a cadena. Para convertir de número a cadena hay que indicar el tipo de variable numérica y tener presente que la longitud de la cadena debe poder guardar la totalidad del número. Admite también los formatos de salida, es decir, que se pueden coger distintas partes del número. La cabecera es **<stdlib.h>**.	sprintf(var_cadena,"identificador",var_numerica);

<table>
<tr>
<td>itoa</td>
<td>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>.
<table>
<tr><th>BASE</th><th>DESCRIPCIÓN</th></tr>
<tr><td>2</td><td>Convierte el valor en binario.</td></tr>
<tr><td>8</td><td>Convierte el valor a Octal.</td></tr>
<tr><td>10</td><td>Convierte el valor a decimal.</td></tr>
<tr><td>16</td><td>Convierte el valor a hexadecimal.</td></tr>
</table>
</td>
<td>itoa(var_entero,var_cadena,base);</td>
</tr>
<tr>
<td>ltoa</td>
<td>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>.</td>
<td>ltoa(var_long,var_cadena,base);</td>
</tr>
</table>

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

- Entrada y salida de datos (stdio.h)
- Memoria dinámica (stdlib.h)

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 PROTOTIPO: tipo_devuelto nombre_funcion ([parámetros]);

SINTAXIS DE LA LLAMADA: nombre_funcion([parámetros]);

SINTAXIS DEL DESARROLLO:

```
tipo_devuelto nombre_funcion
([parámetros])
    {
      cuerpo;
    }
```

Ejercicio

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 esta 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.

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();
```

Ejercicio

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 a 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();

Ejercicio

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);
```

Ejercicio

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, por favor revisa el anexo 9. **(ANEXO 9).**

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.

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()

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, 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 claras y sencillas. 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.

Ejercicio

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));
}
```

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

RESUMEN DE LA UNIDAD

En esta unidad se vieron las características de las funciones internas del lenguaje C, las funciones definidas por el usuario, además del ámbito de las variables y la recursividad.

GLOSARIO

Función

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

Función interna

Función predefinida en un lenguaje de programación.

Función definida por el usuario

Función creada por un programador para un fin específico.

Recursividad

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

Variable local

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

Variable global

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

ACTIVIDADES DE APRENDIZAJE

#	Actividad
1.	Realiza un programa que transforme una cadena introducida por el usuario en minúsculas a su equivalente en mayúsculas.
2.	Realiza un programa en C, que simule una pantalla que pida un login y una contraseña. El programa debe mandar un mensaje si el login y el password son correctos y mandar otro mensaje en caso de que el login y el password sean incorrectos.
3	Realiza una función que busque un número que introduzca el usuario, en un arreglo de 100 números.
4	Realiza una función que determine si un número introducido por el usuario es par o impar.
5	Realiza una función que obtenga el máximo de tres enteros introducidos por el usuario, utiliza variables locales.
6	Realiza una función que calcule el pago de mensualidades para una deuda de $16,000.00 pesos, tomando en cuenta un plazo de un año, y un interés anual de 24%, utiliza variables globales.
7	Realiza una función que obtenga la factorial de un número.

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?

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

LO QUE APRENDÍ

Realiza una función que ordene un arreglo de 100 números utilizando el algoritmo de quicksort.

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

EXAMEN DE AUTOEVALUACIÓN

I. 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 stract 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
- ❍ 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:

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

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

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

8. La palabra reservada coid antes de una función indica:

- ❍ a) Que la función devuelve un valor
- ❍ b) Que la función no devuelve nada
- ❍ c) Que la función devuelve un entero
- ❍ d) Que la función 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:

- ❍ a) Que la función carece de parámetros
- ❍ b) Que la función tiene un parámetro
- ❍ c) Que la función tiene más de un parámetro
- ❍ d) Que 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 *srtcat* 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) Obtiene el logaritmo de un número
- d) Obtiene el coseno de un número

19. Lo primero al crear una función es:

- a) Crear el prototipo
- b) Llamar a la función
- c) Indicar sus parámetros
- d) Determinar su ámbito

20. Una función recursiva es una función que:

- a) Se llama a sí misma
- b) Devuelve valores
- c) No devuelve valores
- d) Usa parámetros

MESOGRAFÍA

BIBLIOGRAFÍA BÁSICA

1. Ceballos, Francisco Javier. ”Lenguaje C”, Alfaomega, 1997, 884 pp.

2. Deitel, H.M., Deitel, P.J. “Como programar en C/C++ y Java”, México: Prentice Hall, 4a Edición, 2004, 1113 pp.

3. Joyanes, Luis. “Fundamentos de Programación”, España: Pearson Prentice Hall, 3a Edición, 2003, 1004 pp.

BIBLIOGRAFÍA COMPLEMENTARIA

1. García, Luis, Juan Cuadrado, Antonio De Amescua y Manuel Velasco, Construcción lógica de programas, Teoría y problemas resueltos, México, coedición Alfa omega-RaMa, 2004, 316 pp.
2. López, Leobardo, Programación estructurada, un enfoque algorítmico, 2ª. Ed., México, Alfa omega, 2004, 664 pp.
3. Sedgewick, Robert, Algoritmos en C++, México, Adisson-Wesley Iberoamericana, 1995, 800 pp.

SITIOS ELECTRÓNICOS

Sitio
www.monografias.com
www.lawebdelprogramador.com

UNIDAD 5

TIPOS DE DATOS COMPUESTOS (ESTRUCTURAS)

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.

LO QUE SÉ

Con tus propias palabras, indica las diferencias más importantes entre un arreglo y una estructura.

Comparte tus respuestas con los demás compañeros en el Foro general de discusión y envíalas a tu asesor.

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];
```

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 como se pueden acceder a los elementos de un arreglo, es de la siguiente forma:

```
listanum[2] = 15; /* Asigna 15 al 3er elemento del arreglo listanum*/
num = listanum[2]; /* Asigna el contenido del 3er elemento a la variable num */
```

El lenguaje C no realiza comprobación de contornos en los arreglos. En el caso de que sobrepase el final durante una operación de asignación, entonces se asignarán valores a otra variable o a un trozo del código, esto es, si se dimensiona un arreglo de tamaño *N*, se puede referenciar el arreglo por encima de *N* sin provocar ningún mensaje de error en tiempo de compilación o ejecución, incluso aunque probablemente se provoque 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];
```

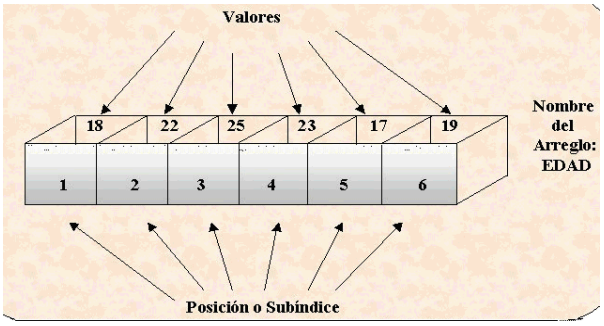
Por ejemplo un arreglo de enteros bidimensionales se escribirá como:

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

Observar que para declarar cada dimensión lleva sus propios paréntesis cuadrados.

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

tabladenums[2][3] = 15; /* Asigna 15 al elemento de la 3ª fila y la 4ª columna*/
num = tabladenums[25][16];



SINTAXIS
tipo nombre_arreglo [nº elementos]; tipo nombre_arreglo [nº elementos]={valor1,valor2,valorN}; tipo nombre_arreglo[]={valor1,valor2,valorN};
INICIALIZACION DE UN ELEMENTO
nombre_arreglo[indice]=valor;
UTILIZACION 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();
}
```

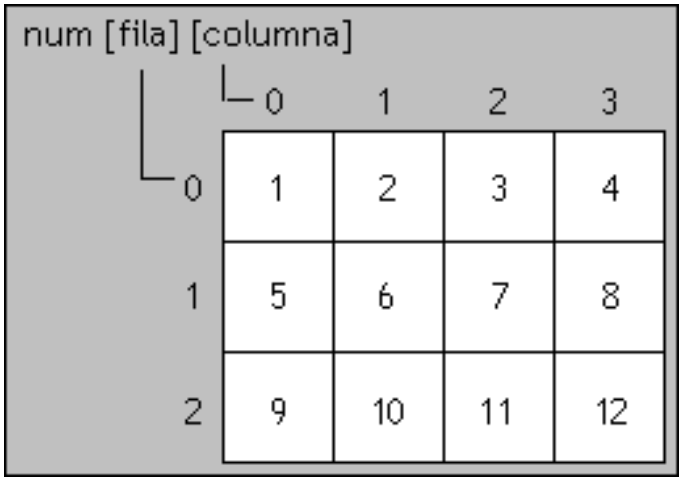
Revisa los **ejercicios de arreglos unidimensionales** para que puedas ver más ejemplos. **(ANEXO 10).**

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.

SINTAXIS:

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



INICIALIZACIÓN DE UN ELEMENTO:

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 (ANEXO 11).

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];
```

Se pueden hacer también inicializaciones de arreglos de caracteres en donde automáticamente C asigna el caracter 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.

En C una estructura es una colección de variables que se referencian bajo el mismo nombre. Una estructura proporciona un medio conveniente para mantener junta información que se relaciona. 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 como sigue:

```
struct direc info_direc;
```

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, seria imposible manejarla en un arreglo, por lo que es necesario utilizar una estructura.

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

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

# include <stdio.h>

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

main()
{
   printf("Introduzca el fabricante del coche.\n");
   gets(miauto.fabricante);
   printf("Introduzca el modelo.\n");
   gets(miauto.modelo);
   printf("Introduzca la matricula.\n");
   gets(miauto.matricula);
   printf("Introduzca la antiguedad.\n");
   scanf("%d",&miauto.antiguedad);
   printf("Introduzca el kilometraje.\n");
   scanf("%ld",&miauto.kilometraje);
   printf("Introduzca el precio.\n");
   scanf("%f",&miauto.precio_nuevo);
```

```
    getchar(); /*vacia la memoria intermedia del teclado*/

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

Por favor revisa el anexode **ejercicios de estructuras** para que puedas ver más ejemplos. **(ANEXO 12).**

RESUMEN DE LA UNIDAD

En esta unidad se verán 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 tratará el tema de los estructuras, una estructura es un conjunto de variables de distinto tipo.

GLOSARIO

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.	Realiza un programa que determine el mayor de tres números almacenados en un arreglo.
2.	Realiza un programa que determine si el contenido de dos arreglos, son iguales.
3	Realiza un programa en C que multiplique dos matrices.
4	Realiza un programa que cuente la cantidad de letras mayúsculas, en un arreglo de caracteres de 100 letras.
5	Realiza un programa que cuente la cantidad de saltos de línea en un arreglo.
6	Realiza un cuadro comparativo que indique las diferencias entre un arreglo y una estructura.
7	Realiza 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.

CUESTIONARIO
DE REFORZAMIENTO

Responde 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 flushall()?

Realiza el cuestionario en un procesador de textos y envíalo a tu asesor.

LO QUE APRENDÍ

Realiza un programa que indique en que posición de un arreglo bidimensional de 5x5 (que tiene almacenados 25 números), se encuentra almacenado el número mayor.

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

EXAMEN DE AUTOEVALUACIÓN

I. 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:

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

6. Un arreglo multidimensional es:

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

7. Para acceder aun 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 antecedido 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 carácter:

- 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 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 datos

21. Una cadena termina con el carácter:

- 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 carácter 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

MESOGRAFÍA

BIBLIOGRAFÍA BÁSICA

1. Cairo, Osvaldo. ”Metodología de la Programación”, México, Alfaomega, 2a Edición, 2003, 438 pp.

2. Ceballos, Francisco Javier. ”Lenguaje C”, Alfaomega, 1997, 884 pp.

3. Deitel, H.M., Deitel, P.J. “Como programar en C/C++ y Java”, México: Prentice Hall, 4a Edición, 2004, 1113 pp.

4. Joyanes, Luis. “Fundamentos de Programación”, España: Pearson Prentice Hall, 3a Edición, 2003, 1004 pp.

BIBLIOGRAFÍA COMPLEMENTARIA

1. López, Leobardo, Programación estructurada, un enfoque algorítmico, 2ª. Ed., México, Alfa omega, 2004, 664 pp.

2. Weiss, Mark Allen, Estructuras de datos en JAVA, México, Addison Wesley, 2000, 740 pp.

SITIOS ELECTRÓNICOS

Sitio	Descripción
www.lawebdelprogramador.com	Documentos de temas de computación
www.monografias.com	Documentos de temas de computación entre otros

UNIDAD 6

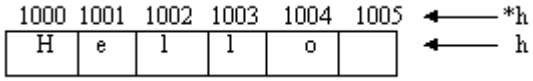
MANEJO DE APUNTADORES

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

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.



C usa los apuntadores en forma extensiva
Es la única forma de expresar algunos cálculos. Se genera código compacto y eficiente. Es una herramienta muy poderosa
C usa apuntadores explícitamente con
Es la única forma de expresar algunos cálculos. Se genera código compacto y eficiente. Es una herramienta muy poderosa.
C usa apuntadores explícitamente con
Arreglos, Estructuras y Funciones

Un apuntador es una *variable* que contiene la dirección en memoria de otra variable. Se pueden tener apuntadores a cualquier tipo de variable.

El operador *unario* o *monádico &* devuelve la dirección de memoria de una variable.
El operador de *indirección* o *dereferencia* * devuelve el “contenido de un objeto apuntado por un apuntador".

Para declarar un apuntador para una variable entera hacer:

```
int *apuntador;
```

LO QUE SÉ

Realiza una breve investigación acerca de los apuntadores, arreglos, estructuras, funciones y la memoria dinámica.

Identifica cuál es la relación entre estos y envíalo a tu asesor.

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).

SINTAXIS:

tipo *nombre;

OPERADORES

El & devuelve la dirección de memoria de su operando. El * devuelve el valor de la variable localizada en la dirección que sigue. Pone en m la dirección de memoria de la variable cuenta. La dirección no tiene nada que ver con el valor de cuenta. En la segunda línea pone el valor de cuenta en q.
 	
m= &cuenta;
 	
q=*m;
 	
Ejercicio
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 utiliza dos apuntadores, en el apuntador p1 se guara la dirección de memoria de la variable x y en la variable p2 se asigna el contenido del apuntador p1, por ultimo se imprime el contenido del apuntador p2.

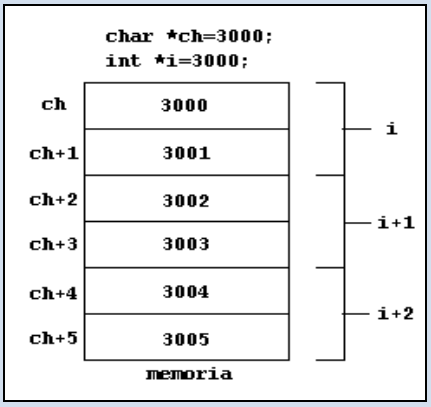
Ejercicio

```
#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;
```



COMPARACION
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.

DECLARACIÓN:	tipo *nombre[nº elementos];
ASIGNACIÓN:	nombre_arreglo[índice]=&variable;
UTILIZAR ELEMENTOS:	*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 nº de Dia: ");
  scanf("%d",&num);
  dias(num);
  getch();
}
void dias(int n)
{
  char *dia[]={"Nº de dia no Valido",
  "Lunes",
  "Martes",
  "Miércoles",
  "Jueves",
  "viernes",
  "Sábado",
```

```
    "Domingo"};
    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 (ANEXO 13)

6.4 Apuntadores y funciones

Cuando C pasa argumentos a funciones, las pasas *por valor*, es decir, si el parámetro es modificado dentro de la función, una vez que termina la función el valor pasado de la variable permanece inalterado.

Hay muchos casos 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().

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

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

#define MAX 256

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

return(0);
}
```

RESUMEN DE LA UNIDAD

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

GLOSARIO

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.
2	Realiza un programa que convierta una cadena en minúsculas a su equivalente en mayúsculas, utiliza la función toupper.
3	Realiza 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.
4	Realiza una función que sume dos números introducidos por el usuario, utiliza apuntadores.
5	Realiza un programa que determine la cantidad de memoria RAM de una computadora.

CUESTIONARIO
DE REFORZAMIENTO

Responde 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?

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

LO QUE APRENDÍ

Realiza un programa que sea capaz de sumar y restar las direcciones de memoria de dos apuntadores, con datos introducidos por el usuario.

Realiza tu actividad en un procesador de textos y envíala a tu asesor.

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 valida, 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. La suma y la resta son operadores _____ que se pueden usar como punteros.

6. Se puede utilizar la _____ para saber que 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 la biblioteca:

- a) string.h para usar los apuntadores
- b) stdio.h se pueden usar apuntadores
- c) conio.h para usar los apuntadores
- d) strdlib.h para usar los apuntadores

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

- a) ´&´
- b) ´*´
- c) ´->´
- d) ´<-´

7. Podemos definir a una estructura como:

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

8. Podemos definir a una estructura como:

- a) Un conjunto de variables de tipos distintos
- b) Un conjunto de variables
- c) Un conjunto de variables del mismo tipo
- d) Un conjunto de 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:

- 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

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) *

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

MESOGRAFÍA

BIBLIOGRAFÍA BÁSICA

1. Ceballos, Francisco Javier. ”Lenguaje C”, Alfaomega, 1997, 884 pp.

2. Deitel, H.M., Deitel, P.J. “Como programar en C/C++ y Java”, México: Prentice Hall, 4a Edición, 2004, 1113 pp.

BIBLIOGRAFÍA COMPLEMENTARIA

1. López, Leobardo, Programación estructurada, un enfoque algorítmico, 2ª. Ed., México, Alfa omega, 2004, 664 pp.

2. Weiss, Mark Allen, Estructuras de datos en JAVA, México, Addison Wesley, 2000, 740 pp.

SITIOS ELECTRÓNICOS

Sitio	Descripción
www.lawebdelprogramador.com	Documentos de temas de computación
www.monografias.com	Documentos de temas de computación entre otros

ANEXO 1:

LENGUAJES DE PROGRAMACIÓN

HISTORIA Y EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

INTRODUCCIÓN

Los ordenadores no hablan nuestro idioma, son maquinas y como tales, necesitan un lenguaje especifico 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 corre

(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 sé hacia 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, voy a redactar 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 esta 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, ordenes y símbolos reconocibles por autómata, a través de su unidad de programación, que le permite ejecutar la secuencia de control deseada. Al conjunto de total de estas instrucciones, ordenes y símbolos que están disponibles se le llamar lenguajes de programación del autómata.

El programa esta 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, calculo/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 ingles, que la principio del siglo XIX predijo muchas de las teorías en que se basan los actuales ordenadores.

Consistía en lo que él denominaba la maquina analítica, pero que por motivos técnicos no pudo construirse hasta mediados del siglo XX. Con él colaboro Ada Lovedby, la cual es considerada como la primera programadora de la historia, pues realizo programas para aquélla supuesta maquina de Babagge, en tarjetas perforadas. Como la maquina no llego nunca 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. A pesar de ello, Ada ha permanecido como la primera programadora de la historia. Se dice por tanto 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 apoyo para crear el proyecto de una máquina de diferencias, un dispositivo mecánico para efectuar sumas repetidas. Pero Babagge se dedico al proyecto de la máquina analítica, abandonando la maquina de diferencias, que se pudiera programar con tarjetas perforadas, gracias a 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 intento crear la máquina que se pudiera programar con tarjetas perforadas para efectuar cualquier cálculo con una precisión de 20 dígitos. Pero la tecnología de la época no bastaba para hacer realidad sus ideas. Si bien las ideas de Babagge no llegaron a materializarse de forma definitiva, su contribución es 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, dado que faltaba algo fundamental: la electrónica. El camino señalado de Babbage, no fue nunca abandonado y siguiéndolo, se construyeron los primeros computadores.

Cuando surgió el primer ordenador, el famoso ENIAC (Electronic Numerical Integrator And Calculator), su programación se basaba en componentes físicos, o sea, que se programaba, cambiando directamente el Hardware de la maquina, exactamente lo que sé hacia era cambiar cables de sitio para conseguir así la programación de la maquina. La entrada y salida de datos se realizaba mediante tarjetas perforadas.

LAS TENDENCIAS DE LOS LENGUAJES DE PROGRAMACIÓN

El estudio de los lenguajes de programación agrupa tres intereses diferentes; el del programador profesional, el del diseñador del lenguaje y del Implementador del lenguaje. Además, estos tres trabajos han de realizarse dentro de las ligaduras y capacidades de la organización de una computadora y de las limitaciones fundamentales de la propia "calculabilidad". El termino "el programador" es un tanto amorfo, en el sentido de que camufla importantes diferencias entre distintos niveles y aplicaciones de la programación. Claramente el programador que ha realizado un curso de doce semanas en COBOL y luego entra en el campo del procesamiento de datos es diferente del programador que escribe un compilador en Pascal, o del programador que diseña un experimento de inteligencia artificial en LISP, o del programador que combina sus rutinas de FORTRAN para resolver un problema de ingeniería complejo, o del programador que desarrolla un sistema operativo multiprocesador en ADA.

En este trabajo, intentare clarificar estas distinciones tratando diferentes lenguajes de programación en el contexto de cada área de aplicación diferente. El "diseñador del lenguaje" es también un termino algo nebuloso. Algunos lenguajes (como APL y LISP) fueron diseñados por una sola persona con un concepto único, mientras que otros (FORTRAN y COBOL) son el producto de desarrollo de varios años realizados por comités de diseño de lenguajes.

El "Implementador del lenguaje" es la persona o grupo que desarrolla un compilador o interprete para un lenguaje sobre una maquina particular o tipos de maquinas. Mas frecuentemente, el primer compilador para el lenguaje Y sobre la maquina X es desarrollada por la corporación que manufactura la maquina X . Por ejemplo, hay varios compiladores de Fortran en uso; uno desarrollado por IBM para una maquina IBM, otro desarrollado por DEC para una maquina DEC, otro por CDC, y así sucesivamente. Las compañías de software también desarrollan compiladores y también lo hacen los grupos de investigación de las universidades.

Por ejemplo, la universidad de Waterloo desarrolla compiladores para FORTRAN Y PASCAL, los cuales son útiles en un entorno de programación de estudiantes debido a su superior capacidad de diagnostico y velocidad de compilación.

Hay también muchos aspectos compartidos entre los programadores, diseñadores de un lenguaje implementadores del mismo. Cada uno debe comprender las necesidades y ligaduras que gobiernan las actividades de los otros dos.

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

Los "Lenguajes Declarativos" son los más parecidos al castellano o ingles en su potencia expresiva y funcionalidad están en el nivel más alto respecto a los otros. Son fundamentalmente lenguajes de ordenes, dominados por sentencias que expresan "Lo que hay que hacer" en ves de "Como hacerlo". Ejemplos de estos lenguajes son 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 más rápidamente el lenguaje y usarlo en su trabajo, sin necesidad de programadores o practicas de programación.

Los lenguajes de " Alto Nivel" son los más utilizados como lenguaje de programación. Aunque no son fundamentalmente declarativos, estos lenguajes permiten que los algoritmos se expresen en un nivel y estilo de escritura fácilmente legible y comprensible por otros programadores. Además, los lenguajes de alto nivel tienen normalmente las características de " Transportabilidad". Es decir, están implementadas sobre varias maquinas de forma que un programa puede ser fácilmente " Transportado " (Transferido) de una maquina a otra sin una revisión sustancial. En ese sentido se llama "Independientes de la maquina". Ejemplos de estos lenguajes de alto nivel son 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).

Los "Lenguajes Ensambladores" y los "Lenguajes Maquina" son dependientes de la maquina. Cada tipo de maquina, tal como VAX de digital, tiene su propio lenguaje maquina distinto y su lenguaje ensamblador asociado. El lenguaje Ensamblador es simplemente una representación simbólica del lenguaje maquina 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 lenguajes.

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

Como muestra este ejemplo, a más bajo nivel de lenguaje mas cerca esta de las características de un tipo e maquina 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 aquí es que los lenguajes ensambladores se utilizan símbolos (X, Y, Z, A para " sumar", M para "multiplicar"), mientras que se requieren códigos numéricos (OC1A4, etc.) para que lo comprenda la maquina.

La programación de un lenguaje de alto nivel o en un lenguaje ensamblador requiere, por tanto, algún tipo de interfaz con el lenguaje maquina para que el programa pueda ejecutarse. Las tres interfaces más comunes: un "ensamblador, un "compilador" y un "interprete". El ensamblador y el compilador traducen el programa a otro equivalente en

el lenguaje X de la maquina "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 maquina. 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 maquina solo una vez.

Algunos lenguajes son lenguajes principalmente interpretados, como APL, PROLOG y LISP. El resto de los lenguajes −− Pascal, FORTRAN, COBOL, PL/I, SNOBOL, C, Ada y Modula−2 − son normalmente lenguajes compilados. En algunos casos, un compilador estará utilizable alternativamente para un lenguaje interpretado (tal como LISP) e inversamente (tal como el interprete SNOBOL4 de los laboratorios Bell). 1 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 diagnosis 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 maquinas cuyos lenguajes son ellos mismos1lenguajes

de alto nivel. Como ejemplo de estos están las nuevas maquinas LISP, las cuales han sido diseñadas recientemente por Symbolics y Xerox Corporations.

Los lenguajes de Programación son tomados de diferentes perspectivas. Es importante para un programador decidir cuales conceptos emitir o cuales 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 y son los siguientes:

- Orientados a Objetos.
- Imperativa: Entrada, procesamiento y salidas de Datos.
- Funcional: "Funciones", los datos son funciones, los resultados pueden ser un valor o una función.
- Lógico: {T, F} + operaciones lógicas (Inteligencia Artificial).
- Concurrente: Aún esta 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 que presentan características diferentes. Por ejemplo, los lenguajes "mas 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 a 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 ejercer las arquitecturas y aplicaciones de las computadoras sobre el diseño de lenguajes y evitar futuros defectos de diseño aprendido 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 diseñado de lenguajes y la mayoría de las aplicaciones con las que se enfrenta el programador. Para los lectores que estén interesados en conocer con más detalle la historia de los lenguajes de programación recomendamos las actas de una recién conferencia (1981) sobre este tema, editadas por Richard Wexelblat.

Vemos que FORTRAN I es un ascendente directo de FORTRAN II, mientras que FORTRAN, COBOL, ALGO 60, LISP, SNOBOL y los lenguajes ensambladores, influyeron en el diseño de PL/I.

También varios lenguajes 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 esta estandarizado, las maquinas que implementan este lenguaje deben cumplir todas las especificaciones estándares, reforzando así el máximo de transportabilidad de programas de una maquina a otra. La policía federal de no comprar maquinas que no cumplan la versión estándar de cualquier lenguaje que soporte tiende a "fortalecer" el proceso de estandarización, puesto que el gobierno es, con mucho, el mayor comprador de computadoras de la nación.

Finalmente, la notación algebraica ordinaria, por ejemplo, influyo fuertemente en el diseño de FORTRAN y ALGOL. Por otra parte, el inglés influyo en el desarrollo del COBOL. El lambda cálculo de Church dio los fundamentos de la notación funcional de LISP, mientras que el algoritmo de Markov motivo el estilo de reconocimiento de formas de SNOBOL. La arquitectura de computadoras de Von Neumann, la cual fue una evolución de la maquina más antigua de Turing, es el modelo básico de la mayoría de los diseños de computadoras de las ultimas tres décadas. Esta maquina no solo influyeron en los primeros lenguajes sino que también suministraron el esqueleto operacional sobre el que evoluciono la mayoría de la programación de sistemas.

Una discusión más directa de todos estos primeros modelos no está entre los objetivos de este texto. Sin embargo, es importante apuntar aquí debido a su fundamental influencia en la evolución de los primeros lenguajes de programación, por una parte, y por su estado en el núcleo de la teoría de la computadora, por otra. Mas sobre este punto, cualquier

algoritmo que pueda describirse en ingles o castellano puede escribirse igualmente como una maquina de Turing (maquina 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:

- ✓ Primera generación. Lenguajes maquina y ensambladores.
- ✓ Segunda generación. Primeros lenguajes de alto nivel imperativo (FROTRAN, COBOL).
- ✓ Tercera generación. Lenguajes de alto nivel imperativo. Son los mas utilizados y siguen vigentes en la actualidad (ALGOL 8, PL/I, PASCAL, MODULA).·
- ✓ Cuarta generación. Orientados básicamente a las aplicaciones de gestión y al manejo de bases de datos (NATURAL, SQL).
- ✓ Quinta generación. Orientados a la inteligencia artificial y al procesamiento de los lenguajes naturales (LISP, PROLOG).

Para la mejor compresión se harán unas definiciones:

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

Código fuente: esta 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.

Tabla. Evolucion de los lenguajes de programación

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

	flexibles y generales	
1971 − 75	Micro ordenadores	
	Sistemas de almacenamiento	PASCAL
	masivo de datos pequeños	COBOL 74
	y baratos	PL /I
	Progr. Estructurada	
	Ingeniería del software	
	Leng. sencillos	
1976 − 80	Comp. baratas y potentes	ADA
	Sistemas distribuidos	FORTRAN 77
	Prog. tiempo−real	PROLOG
	Prog. interactiva	C
	Abstracción de datos	
	Prog. con fiabilidad	
	y fácil mantenimiento	

Todo este desarrollo de las computadoras y de los lenguajes de programación, suele divisarse por generaciones y el criterio que se determinó para determinar el cambio de generación no está muy bien definido, 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, acceso transparente a datos y capacidad de trabajo. En red, lo hacen más complejo aun. 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.

- *Métodos*

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

- *Mensajes*

  Es simplemente una petición de un objeto a otro para que este 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 la conoce como paso de mensajes.

- *Clases*

  Es un tipo 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 mas 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 mas 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 modulo nuevo (extensibilidad) sin afectar al resto de la aplicación.

6) Lenguajes en POO.

- *Puros*

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

- *Híbridos*

  Son los que permiten la POO con la programación estructurada. Ej: 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:

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.

Pozos o fuentes de datos: estas clases generan datos o los aceptan para procesarlos mas adelante. A diferencia de los anteriores, estas clases no retiene los datos por un periodo de tiempo sino que los genera sobre demanda o los procesa cuando se le llama.

Vistas: se encargan de la presentación de la información.

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 interface1, [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 el; 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 este.

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]
[syncronized] tipodato_devuelto
                                nombre_metodo (parametros)[throws
exepcion1 [,exepcion2]]
                                        {
                                                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.

Syncronized

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 linea (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 explicita).

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 mas 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 ocupo 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 parte 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 sobrecarga 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 valido 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 mas 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 mas reducido o esta mas 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 do 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 muevas 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 mas 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

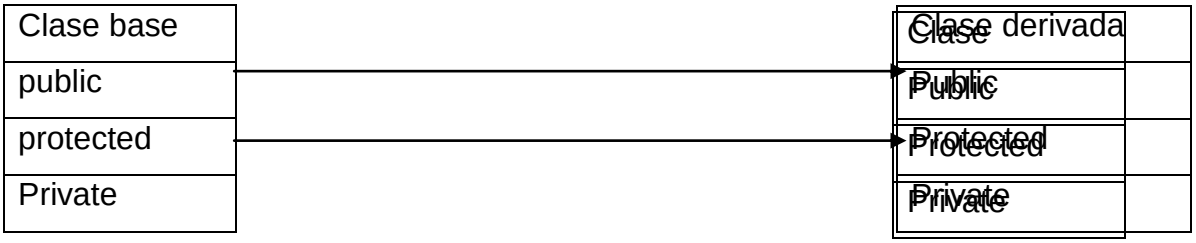
C++:

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

Si en la derivación especificamos:

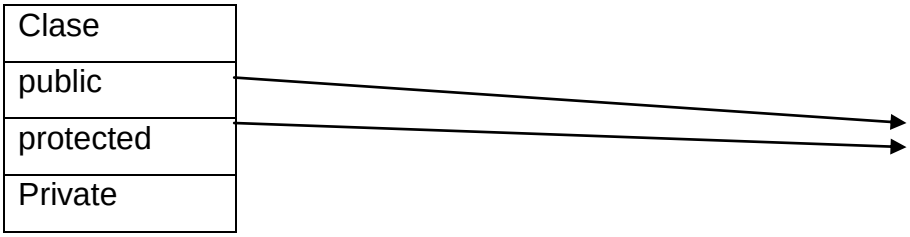
♦ Public

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.



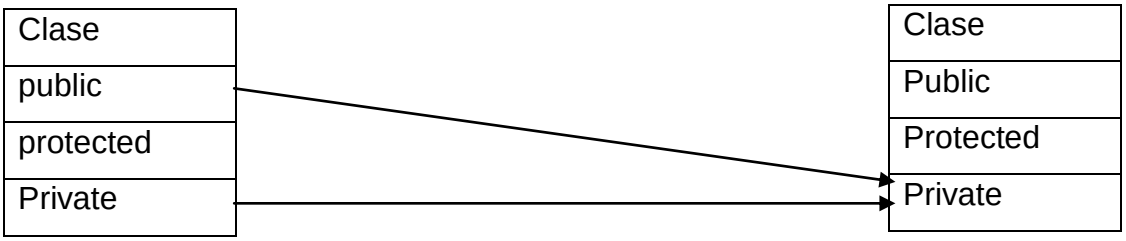
♦ Protected

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 explicita.

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 ultimo, 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.

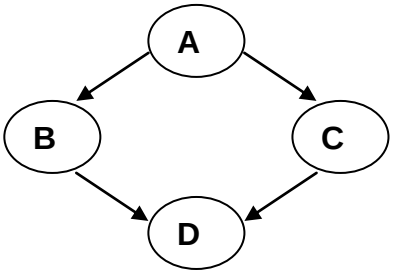
```
Class hija extends madre
    {
        public hija (tipo1 x)
        {
            super (y);
            cuerpo del constructor;
        ]
    }
```

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, la cual tiene sus propios miembros. Definimos dos clases derivadas de esta 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 heredara las funciones miembro de b y c, pero también derivara 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 que 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 esta referenciando. En este caso se produce una ligadura dinámica.

→ Con funciones virtuales (c++)

Cuando varias sub-clases derivan de una clase base, estas 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:

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

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

La función debe ser declarada como virtual en la primer clase en la que esta 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 esta 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.

ANEXO3

OPERADORES LÓGICOS Y RELACIONALES. EJERCICIOS

EJERCICIO 1

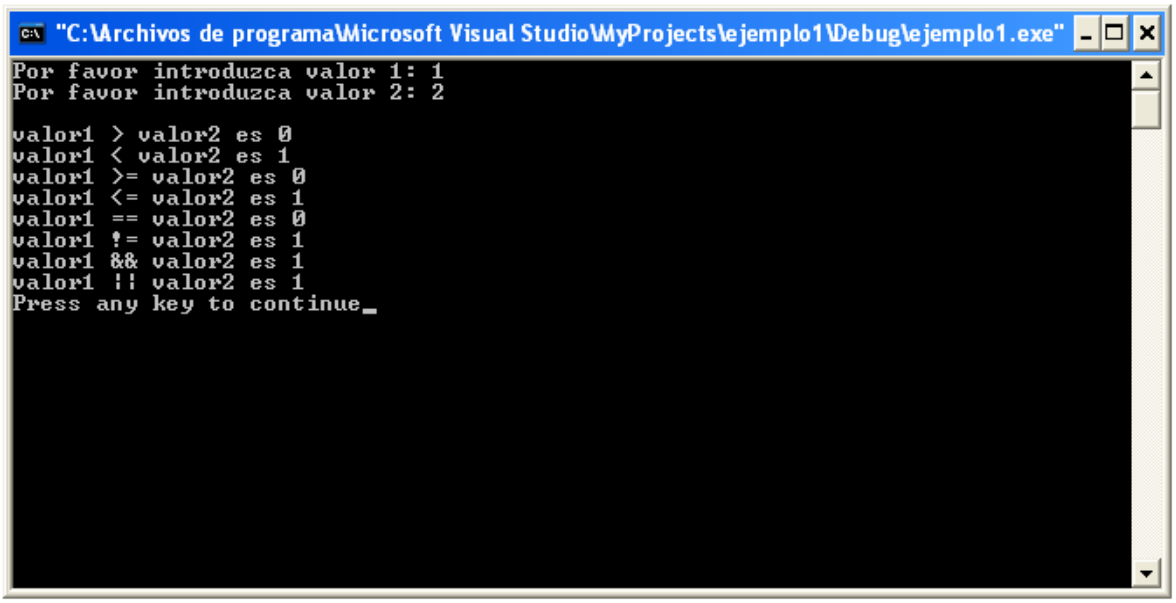
El siguiente ejercicio muestra el uso de los operadores lógicos y relacionales. El usuario debe introducir dos valores numéricos y el programa hace comparaciones relaciones y lógicas con los valores introducidos.

```
/*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:

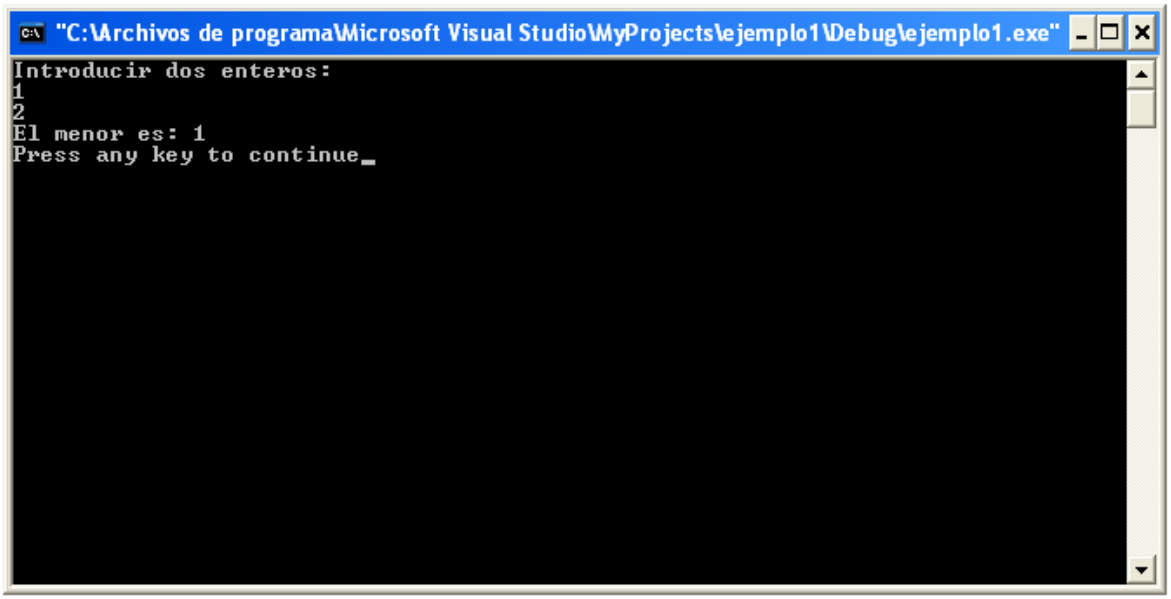


Veamos un programa que indica el menor de dos números leídos.

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

ANEXO 4

ESTRUCTURA SECUENCIAL

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);
}
```

ANEXO 5

ESTRUCTURA ALTERNATIVA O CONDICIONAL

EJERCICIO 1

Una persona desea saber cuánto dinero se genera por concepto de intereses sobre la cantidad que tiene en inversión en el banco. Él decidirá reinvertir los intereses siempre y cuando sean iguales o mayores a $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 a 70 puntos; y entregó un trabajo final. (No se toma en cuenta la calificación del trabajo final, sólo se toma en cuenta si lo entregó).

```
# include <stdio.h>
main()
{
   float examen1, examen2, examen3,final;
   int tra_fin=0;

   printf("Por favor introduzca la calificacion del primer examen: ");
   scanf("%f",&examen1);
   printf("Por favor introduzca la calificacion del segundo examen: ");
   scanf("%f",&examen2);
   printf("Por favor introduzca la calificacion del tercer examen: ");
   scanf("%f",&examen3);
   printf("Introduzca 1 si entrego 0 si no entrego trabajo final: ");
   scanf("%d",&tra_fin);
```

```
  final = (examen1+examen2+examen3)/3;
  if(final < 7 || tra_fin == 0)
    printf("Tendras que cursar programacion nuevamente\n");
  else
    printf("Aprobaste con la siguiente calificacion: %.2f\n",final);

  return(0);
}
```

EJERCICIO 3

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

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

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

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

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

EJERCICIO 4

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

```
#include <stdio.h>

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

  switch(opcion)
  {
    case 1:
```

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

ANEXO 6

ESTRUCTURA REPETITIVA
EJERCICIOS DO WHILE

EJERCICIO 1

Realiza un programa que lea una 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, 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));
        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");
             }
    }
}
```

ANEXO 7

ESTRUCTURA REPETITIVA EJERCICIOS FOR

EJERCICIO 1

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

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

EJERCICIO 2

Lee 10 números y obtén su cubo y su cuarta.

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

EJERCICIO 3

Lee 10 números e imprime solamente los 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);
        }
}
```

ANEXO 8

FUNCIONES DE CARACTERES Y CADENAS

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();
```

ANEXO 9

FUNCIONES DEFINIDAS POR EL USUARIO EJERCICIOS

EJERCICIO 1

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

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

```
                printf ("El numero %d no es impar. \n", n);
}

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

EJERCICIO 2

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

```
#include<stdio.h>

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

main()
{
    int a,b,c,d;
    printf("Introduce el valor de a: ");
      scanf("%d",&a);
    printf("\nIntroduce el valor de b: ");
```

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

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

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

EJERCICIO 3

Realiza una función que eleve un número al cubo.

```
#include<stdio.h>

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

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

EJERCICIO 4

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

```
#include<stdio.h>

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

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

EJERCICIO 5

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

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

void impresion(void);

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

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

EJERCICIO 6

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

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

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

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

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

ANEXO 10

ARREGLOS Y ESTRUCTURAS EJERCICIOS

DE ARREGLOS UNIDIMENSIONALES

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};

        static int resul[5]={0,0,0,0,0};

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

```
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]);
}
```

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;
}
```

ANEXO 11

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

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]);
```

ANEXO 12

ARREGLOS Y ESTRUCTURAS

Ejercicios de estructuras

EJERCICIO 1

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

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

# include <stdio.h>

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

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

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

```
  flushall();
  }
  return(0);
}
```

EJERCICIO 2

Veamos ahora un ejemplo con arreglos, cadenas y estructuras.

```
/* Inicialización y manejo de "arreglos", cadenas y estructuras.*/
# include <stdio.h>
void main()
{
int i, j;
static int enteros [5] = { 3, 7, 1, 5, 2 };
static char cadena1 [16] = "cadena";
static char cadena2 [16] = { 'c','a','d','e','n','a','\0' };
static int a[2][5] = {
{ 1, 22, 333, 4444, 55555 },
{ 5, 4, 3, 2, 1 }
};
static int b[2][5] = { 1,22,333,4444,55555,5,4,3,2,1 };
static char *c = "cadena";
static struct {
int i;
float x;
} sta = { 1, 3.1415e4}, stb = { 2, 1.5e4 };
static struct {
char c;
```

```
int i;
float s;
} st [2][3] = {
{{ 'a', 1, 3e3 }, { 'b', 2, 4e2 }, { 'c', 3, 5e3 }},
{ { 'd', 4, 6e2 },}
};
printf ("enteros:\n");
for ( i=0; i<5; ++i ) printf ("%d ", enteros[i]);
printf ("\n\n");
printf ("cadena1:\n");
printf ("%s\n\n", cadena1);
printf ("cadena2:\n");
printf ("%s\n\n", cadena2);
printf ("a:\n");
for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d ", a[i][j]);
printf("\n\n");
printf ("b:\n");
for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d ", b[i][j]);
printf ("\n\n");
printf ("c:\n");
printf ("%s\n\n", c);
printf ("sta:\n");
printf ("%d %f \n\n", sta.i, sta.x);
printf ("st:\n");
for (i=0; i<2; ++i) for (j=0; j<3; ++j)
printf ("%c %d %f\n", st[i][j].c, st[i][j].i, st[i][j].s);
}
```

La salida del anterior programa es la siguiente:

```
enteros:
3 7 1 5 2
cadena1:
cadena
cadena2:
cadena
a:
1 22 333 4444 55555 5 4 3 2 1
b:
1 22 333 4444 55555 5 4 3 2 1
c:
cadena
sta:
1 31415.000000
st:
a 1 3000.000000
b 2 400.000000
c 3 5000.000000
d 4 600.000000
0 0.000000
0 0.000000
```

ANEXO 13

EJERCICIO

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;
ptr = &mi_empleado;/* sintaxis para modificar un miembro de una estructura a través de un apuntador */
ptr->teléfono = 5632332;
strcpy(ptr->nombre, "jonas");
```

```
mostrar_nombre(ptr);
return 0;
}
void mostrar_nombre(empleado *x)
{
/* se debe de usar -> para referenciar a los miembros de la estructura */
printf("nombre del empleado x %s \n", x->nombre);
}
```

La estructura se pasa por referencia:

```
mostrar_nombre(ptr);
```

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

```
empleado *x = ptr;
```

UNIDAD 1					
I	II	III	IV	V	
1. V	1. F	1. C	1. F	1. D	6. A
2. V	2. F	2. E	2. V	2. A	7. C
3. V	3. V	3. D	3. F	3. A	8. D
4. V	4. F	4. B	4. F	4. C	9. A
5. F	5. F	5. 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. B	19. A
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

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