18

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

PROGRAMACION EN C PARA ESTUDIANTES DE ACTUARIA

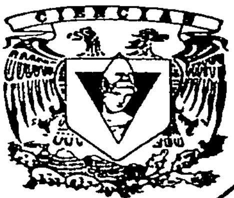
T E S I S

QUE PARA OBTENER EL TITULO DE

A C T U A R I O

P R E S E N T A :

ALEJANDRO CARRILLO NOLAZCO



DIRECTOR DE TESIS: M. en C. MIGUEL MURGUIA ROMERO

TESIS CON FALLA DE ORIGEN

DIVISION DE ESTUDIOS PROFESIONALES
2002
FACULTAD DE CIENCIAS
SECCION ESCOLAR

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

DRA. MARÍA DE LOURDES ESTEVA PERALTA
Jefa de la División de Estudios Profesionales de la Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:
Programación en C para estudiantes de Actuaría.

realizado por Alejandro Carrillo Nolazco

con número de cuenta 9429200-0, quien cubrió los créditos de la carrera de: Actuaría.

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M. en C. Miguel Murguía Romero
Propietario	Dra. Amparo López Gaona
Propietario	M. en C. María Guadalupe Elena Ibargüengoitia Gónzalez
Suplente	Act. Ricardo Sevilla Aguilar
Suplente	Act. Daniel Alejandro Cervantes Cabrera

Consejo Departamental de Matemáticas

M. en C. José Antonio Flores Díaz

MATEMATICAS

A Dios

por estar presente

en cada una de las

personas que

contribuyeron a

realizar mi sueño.

Índice

Introducción 1

Objetivos 3

Justificación y Metodología 4

Capítulo 1
Análisis de planes de Estudio

1.1. Universidades 6
1.2. Comparación de planes de Estudio 6
1.3. Otras Consideraciones 8

Capítulo 2
Introducción a C

2.1. Marco histórico 16
2.2. Características del lenguaje 17
2.3. Compiladores frente a intérpretes 22

Capítulo 3
Conceptos Básicos

3.1. Identificadores 27
3.2. Tipos de datos básicos 27

3.3. printf() 31
3.4. scanf() 41
3.5. Variables 46
3.6. Cualificadores de acceso 49
3.7. Operadores 52
3.8. Expresiones 61
3.9. Moldes 61

Capítulo 4
Condicionales y Ciclos

4.1. Instrucciones de bloque 62
4.2. Verdadero y falso en C 63
4.3. Instrucciones de selección 63
4.4. La alternativa ? 69
4.5. La expresión condicional 69
4.6. switch() 69
4.7. Instrucciones de iteración 77
4.8. El ciclo while() 82
4.9. El ciclo do-while(). 87
4.10. Instrucciones de salto 90

Capítulo 5
Datos Estructurados

5.1. Arreglos y Cadenas 97
5.2. Apuntadores 110
5.3. Funciones 126

5.4. Funciones de tipo void 140
5.5. Lo que devuelve main() 140
5.6. Recursión 140
5.7. Prototipos de funciones 142
5.8. Declaración de listas de parámetros de longitud variable 145
5.9. La regla del entero por default 145

Capítulo 6
C con aplicación Actuarial

6.1. Álgebra Superior 147
6.2. Álgebra Lineal 151
6.3. Geometría Analítica 155
6.4. Matemáticas Financieras 158
6.5. Calculo Integral 160
6.6. Análisis Numérico 161
6.7. Análisis Matemático 163
6.8. Calculo Actuarial 164
6.9. Probabilidad y Estadística 167

Conclusiones 181

Apéndice 1. Cómo usar un compilador de C 182

Apéndice 2. Tabla de Programas 187

Referencias 191

Introducción

La carrera de Actuaría nace en México en el año de 1946 para formar profesionales altamente capacitados para cuantificar el riesgo en la industria del seguro la cual fundamenta sus servicios en la administración financiera de diferentes riesgos.

Sin embargo, aunque la carrera surgió básicamente por los requerimientos y necesidades de las compañías de seguros; con el paso del tiempo los actuarios han ido demostrando que pueden desenvolverse en un elevado nivel profesional no sólo en el área de los seguros, sino también en otros campos, los cuales son hoy en día también propios de la carrera, entre los que destacan demografía, econometría, estadística, finanzas, investigación de operaciones y computación.

En la actualidad puede preverse que el campo de trabajo actuarial seguirá aumentando en los próximos años. La creciente demanda de métodos informáticos específicos para el procesamiento de información, lleva a pensar que también este rubro habrá de constituirse como un área de oportunidad para los actuarios siempre y cuando se encuentren bien preparados para ello. Por eso es importante preparar a las futuras generaciones de Actuarios para que más rápidamente puedan incorporarse a este campo, ya que finalmente el área Computacional proporciona la herramienta de software que requiere el Actuario para la optimización del manejo de la información.

Ante esta situación he elaborado el presente trabajo con la finalidad de poder colaborar en el fortalecimiento y enriquecimiento del área Informática y así contribuir a una mejor formación académica de un Actuario en la parte de la Programación.

Este trabajo fundamenta sus bases en una investigación realizada en cinco universidades que imparten la carrera de Actuaría, donde se pretende mostrar la importancia que tiene la Programación como parte integral en el desarrollo profesional del Actuario.

En este trabajo se presenta un análisis sobre cuál es la postura de las Universidades respecto a las materias de computación, si las incluyen en sus planes y qué importancia les dan.

La parte central de este trabajo es un manual de C; enfocado al ámbito Actuarial, donde se realizan ejemplos, se dan posibilidades y algunas alternativas de campos sobre los que se puede desarrollar software, todo esto basado en un estudio cuidadoso de las materias que llevan en común los seis planes de estudio[1] de las Universidades que se analizan y una investigación en Internet sobre las

[1] La UNAM cuenta con 2 planes de estudio diferentes.

distintas formas de la enseñanza del Lenguaje C. Todo esto para tratar de incorporar los mayores medios posibles para facilitar la enseñanza.

Se deja todo este material en un disco compacto para que pueda ser consultado y utilizado sin tener la necesidad de fotocopiarlo, y con la intenciòn de que la Facultad incorpore ésta y otras tesis a sus páginas de Internet.

En el primer capítulo se realiza un breve análisis de los planes de estudio de la carrera de Actuaría.

En el segundo capítulo se da una breve historia del origen del lenguaje C, algunas de sus características, utilidades y datos importantes acerca de éste.

En el tercer capítulo se proporcionan algunos conceptos básicos para poder empezar a programar en C, tales como: los tipos de datos, variables, parámetros, constantes, operadores, primeros comandos e instrucciones básicas, entre otros.

En el cuarto capítulo se incluyen estructuras como if(), switch(), for(), while() y do-while(); las cuales son utilizadas no solamente en C sino en casi todos los lenguajes de programación estructurada que existen; por lo que he tratado de profundizar lo más posible en estas estructuras para lograr que se dominen por completo, ya que son la base de cualquier programador eficiente.

En el quinto capítulo proporciona material como: arreglos, cadenas, apuntadores y funciones. Lo que en programación llamamos arreglos, en Matemáticas son las famosas matrices; algunos maestros en materias como Análisis Numérico nos enseñan a programar en Matlab que es un software de matrices. Por lo tanto me he esmerado para que este concepto quede bien claro y lo que se aprenda en C pueda ser fácilmente transportado a Matlab. En lo relacionado a Cadenas, Apuntadores y Funciones son un complemento necesario para que podamos programar cosas más interesantes.

El sexto capítulo proporciona una serie de ejemplos sencillos e ilustrativos de Aplicación Actuarial y Matemática[2] que muestran algo de lo que se puede realizar en determinadas áreas, así como despertar el interés de programar cosas mucho más complejas.

[2] Incluyo ejemplos de Matemáticas porque la carrera de Actuaría fundamenta sus bases en estás.

Objetivos

Objetivo General

- ✓ Proporcionar a estudiantes de la carrera de Actuaría un material que cubra sus necesidades, muestre la importancia y ~~necesidad~~aplicación de la programación para éstos.

Objetivos Específicos

- ✓ Mostrar que la Programación es indispensable y necesaria para un Actuario.
- ✓ Dejar abiertos algunos campos para la construcción del software para las materias Actuariales.
- ✓ Hacer de la enseñanza de la programación algo fácil y útil.
- ✓ Dejar las bases de programación para materias que cursa un Actuario en el transcurso de su carrera.
- ✓ Dejar el material en un medio magnético como es el caso de un disco compacto y de esta manera poder incorporar este último medio a las tesis de la Universidad.

Justificación y Metodología

En mi paso por la Facultad de Ciencias me di cuenta que era necesario y útil saber programar; ya que esto podía facilitar mucho las cosas. Por lo anterior empecé a involucrarme un poquito más en esto. Tuve la suerte de estar en la discusión de los planes de Estudio de la Carrera de Actuaría 2000 en la UNAM, y con gusto vi la inclusión de más materias en este campo. Poco después tome la iniciativa de realizar mi servicio social como ayudante de la materia de Programación I; en este lapso de tiempo me dediqué a buscar material que pudiera combinar la programación con la Actuaría y desafortunadamente no encontré ninguno. No solo eso, vi con tristeza que casi no se fomenta el desarrollo de software actuarial, no se explota esa capacidad de análisis y síntesis que tiene la gente que estudia esta carrera, no se siembra la semillita de la programación en los estudiantes.

Al término de mis estudios, ya en búsqueda de trabajo, me di cuenta que era necesario tener conocimientos de programación para poder aspirar a algunos empleos y hoy en día en mi trabajo combino el conocimiento actuarial con el desarrollo de software en este campo.

Por lo mencionado anteriormente, he querido colaborar con un texto que sirva para cubrir las necesidades básicas de todos aquellos que llevan materias de programación (muy en especial a los estudiantes de Actuaría de la Facultad de Ciencias) o que están interesados en saber más de esto. Me di a la tarea de realizar este trabajo de investigación tratando de empujar a los actuarios a un mejor desarrollo profesional y sembrar una semillita de todo lo que podemos lograr en este campo.

Para la realización de este trabajo investigué en varias fuentes e hice uso de varios recursos, comenzando en Internet; aquí revise primero cómo estaba la enseñanza de la carrera de Actuaría en las universidades del país, revisé sus páginas y pedí información extra vía mail a cada una de ellas. Posteriormente realicé un análisis de los diferentes planes de estudio; compraré la temática de las materias de computación, tanto como me lo permitieron las universidades ya que algunas de ellas dan libertad de cátedra.

A continuación realicé una consulta donde me informé sobre las diferentes maneras que hoy en día existen para aprender a programar; traté de tomar lo que a mi consideración era mejor de cada una de estas para así poder realizar una enseñanza que pueda cubrir el mayor número de opciones de aprendizaje.

Escogí como lenguaje de programación a C porque considero que es un buen lenguaje para aprender a programar; es de nivel medio, es estructurado, tiene un excelente compilador, es portable, utiliza bibliotecas, se pueden realizar en él construcciones algorítmicas, contiene estructura de datos, acepta comentarios, es

útil en varios campos y porque es un lenguaje para programadores (en el capítulo 1 del manual profundizo más en estas características).

Posterior a esto, eché un vistazo a las bibliotecas y librerías buscando libros de C, para saber qué temas incluyen, en qué nivel y qué tan extenso son. Además de buscar si existía alguno con enfoque o aplicaciones actuariales.

Mas adelante revise a fondo los programas de las materias de Programación I y II que se imparte en la Facultad de Ciencias de la UNAM, de donde tome algunos aspectos que consideré relevantes incluir en el presente trabajo, obtuve los planes de estudio más representativos de otras universidades para así poder hacer un verdadero análisis de la carrera de Actuaría; por lo cual los analicé y agrupé priorizando de acuerdo a los objetivos que perseguía.

Lo anterior me sirvió para realizar los programas con aplicación actuarial y matemática que incluyo principalmente en el capítulo 6. Así logré que estos tuvieran variedad y representabilidad de cada una de las materias que estos planes tienen en común y así poder proporcionar algunas ideas de lo que se puede realizar en cada una de estas.

Otro aspecto del cual eche mano fue de mi experiencia como ayudante en la materia de Programación I; durante la cual me di a la tarea de observar a los alumnos, dónde y porqué tenían dificultades en el aprendizaje de la programación. Esta experiencia fue muy útil porque me dejó un panorama más amplio de qué aspectos hay que atacar para poder hacer más fácil, divertida y ágil esta enseñanza.

Capítulo 1
Análisis de planes de estudio.

El análisis de los Planes de Estudio fue parte fundamental del por qué y del cómo se diseño este trabajo, por lo que se dedica este apartado a su análisis.

1.1. Universidades

Las Universidades que están consideradas para este análisis son:

- ✓ La UNAM (Universidad Nacional Autónoma de México)
- ✓ El ITAM (Instituto Tecnológico Autónomo de México)
- ✓ La UAG (Universidad Autónoma de Guadalajara)
- ✓ La UAS (Universidad Anáhuac del Sur)
- ✓ La UDLA (Universidad de las Américas Puebla)

Estas universidades fueron escogidas por ser las primeras en impartir la carrera de Actuaría en México. En 1946 la Universidad Nacional Autónoma de México (UNAM) abre por primera vez en México, la Carrera de Actuaría, siendo la primera institución en impartir este programa, seguida por la Universidad Anáhuac, ENEP Acatlán, Universidad Anáhuac del Sur, Instituto Tecnológico Autónomo de México, Universidad de las Américas-Puebla y la Universidad Autónoma de Guadalajara.

1.2. Comparación de los planes de Estudio.

La comparación de los planes de estudio se realizará dividiendo las materias en dos grandes bloques; el primero abarca las materias relacionadas con la computación, programación e informática y el segundo, es el complemento de las mismas.

1.2.1. Materias en común

La **Tabla 1.2.** muestra la comparación de los diferentes planes de estudio que analizaremos a continuación.

Aquí podemos observar que todos los planes de estudio tienen las siguientes materias en común:

- ✓ Cálculo diferencial e Integral
- ✓ Álgebra Superior
- ✓ Geometría Analítica
- ✓ Álgebra Lineal
- ✓ Matemáticas Financieras
- ✓ Probabilidad
- ✓ Estadística
- ✓ Ecuaciones Diferenciales
- ✓ Investigación de Operaciones
- ✓ Contabilidad
- ✓ Seguros de Vida
- ✓ Cálculo Actuarial

Aunque algunas materias no lleven el mismo nombre, su contenido es similar, como lo es el caso del Cálculo Diferencial e Integral, en donde la UDLA lo llama únicamente Cálculo o en el caso de las Matemática Financieras donde la UAS la llama teoría del interés; por mencionar algunos ejemplos.

Existen otras materias como Análisis Matemático que sólo es impartida por tres planes de estudio de manera obligatoria (ENEP Acatlán, Facultad de Ciencias UNAM y UAG) y por uno más de manera opcional (UAS). En las otras dos Universidades (el ITAM y la UDLA) se tiene implícito en la materia de Cálculo Diferencial e Integral. Podemos encontrar muchos casos de estos pero en general se puede ver que la carrera de Actuaría, en esencia, es la misma para todos los planes. Veamos algunas de sus diferencias:

- ✓ El Plan de Estudios del ITAM es un plan muy cuadrado ya que no tiene asignaturas optativas ni bloques de especialidad en algún campo (i.e. todos sus egresados cursan las mismas materias). Además de ser un plan muy cargado a la parte de Seguros y Teoría Actuarial.
- ✓ Los planes de estudios sin áreas de especialización son los de la UAG y la UAS los cuales sólo permiten escoger entre 4 y 8 materias optativas respectivamente.
- ✓ La UDLA tiene bien definidos sus campos de especialización pero no deja abierta la posibilidad para cursar alguna materia optativa.
- ✓ Los planes de estudio de la UNAM son los únicos que te permiten la opción de tomar un bloque de especialización y materias optativas.

1.2.2. Materias de Computación.

A continuación se analiza el bloque de las materias de computación.

Las **Tablas 1.1.** y **1.2.** nos muestran la distribución de materias optativas y obligatorias de computación.

Tabla 1.1.
Materias obligatorias y optativas de computación.

UNIVERSIDAD	OBLIGATORIAS	OPTATIVAS
UAG	6	2
UAS	5	0
Acatlán	4	0
Ciencias	3	5
UDLA	3	3
ITAM	3	0

Con la tabla anterior y la **Tabla 1.2.** podemos percatarnos de lo siguiente:

- ✓ Todas las universidades al menos imparten 3 de estas materias como obligatorias, Ciencias y la UDLA dan la posibilidad de tomar un bloque de especialización y UAG nos permite tomar 2 optativas.
- ✓ Para Acatlán, el ITAM y la UAS es importante impartir una de estas materias en el primer semestre; aunque todas las universidades imparten al menos una en el primer año.
- ✓ Parecería que únicamente en cuatro de los seis planes se lleva al menos una materia de programación y en los otros dos no. Pero esto no es cierto, para el caso de Acatlán se enseña en todas sus materias de computación y para la UAS va implícita la enseñanza en sus materias de Computación I, II y III.

1.3. Otras consideraciones.

Como se habrá advertido, los planes de Actuaría en esencia tienen las mismas bases, bases sobre las cuales se fundamenta en gran parte este trabajo.

Cabe señalar que:

- ✓ No todas las Universidades imparten C como lenguaje de programación, en algunas se da la libertad de cátedra a los profesores que imparten la asignatura.
- ✓ Se considera en el bloque de materias de computación Análisis Numérico[3] o su equivalente porque muchas veces en esta materia se enseña a programar (Como mencione en un principio generalmente en Matlab).
- ✓ En algunas Universidades se enseña el manejo de Bases de Datos, donde también se comienza a programar y se aplican algunos de los conocimientos previos como estructuras de control, ciclos, declaración y tipos de variables.

Por lo mencionado anteriormente he elaborado los siguientes capítulos; los cuales dedico a la enseñanza del Lenguaje C, en donde he procurado que los ejemplos que se realicen sean sencillos de entender y estén enfocados a alguna aplicación actuarial.

[3] También llamado Métodos Numéricos

Tabla 1.2.
Planes de Estudio de la carrera de Actuaría
Asignaturas Obligatorias

ACATLÁN	CIENCIAS	ITAM	UAG	UAS	UDLA
PRIMER SEMESTRE					
Cálculo Diferencial e Integral I	Cálculo Diferencial e Integral I	Introducción a la Matemática Superior	Álgebra Universitaria	Matemáticas Avanzadas	Introduccion a la Actuaria
Álgebra Superior	Algebra Superior I	Herramientas Computacionales y Algoritmos	Calculo Diferencial e Integral I	ÁlgebraSuperior	Estudios Generales I (Ingeniería)
Metodos y Técnicas de Estudio I	Geometría Analítica I	Economía I	Matemáticas Financieras I	Geometría Analítica	Geometría Analitica
Introducción a la Computación	Matemáticas Financieras	Contabilidad I	Geometría Analitica	Teoría del Interés I	Matemática Básica
Seguro de Vida	Problemas Sociales y Economicos de México	Ideas e Instituciones Politicas y Sociales I	Lógica Matemática	Contabilidad	Primer Idioma I
				Computación I	Segundo Idioma I
				Fundamentos de la Ciencia Actuarial	
SEGUNDO SEMESTRE					
Cálculo Diferencial e Integral II	Cálculo Diferencial e Integral II	Calculo Diferencial e Integral I	Álgebra Lineal I	Cálculo Diferencial e Integral I	Matemáticas Financieras I
Geometría Analitica	Álgebra Superior II	Geometría Analítica I	Cálculo Diferencial e Integral II	Álgebra Lineal	Programación Básica
Álgebra Lineal I	Geometría Analitica II	Álgebra Superior I	Introducción a la Computación	Teoria del Interes II	Laboratorio Programación Básica
Estructura y Procesamiento de Datos	Contabilidad	Ideas e Instituciones Políticas y Sociales II	Matemáticas Financieras II	Analisis de Estados Financieros	Teoría de Ecuaciones
Matemáticas Financieras I	Programación I	Problemas de la Civilización Contemporánea I	Seguro de Personas	Computación II	Cálculo I
		Economía II		Seguros I	Primer Idioma II
					Segundo Idioma II
TERCER SEMESTRE					
Matemáticas Financieras II	Cálculo Diferencial e Integral III	Cálculo Diferencial e Integral II	Álgebra Lineal II	Cálculo Diferencial e Integral II	Matemáticas Financieras II
Cálculo Diferencial e Integral III	Álgebra Lineal I	Geometría Analítica II	Cálculo Diferencial e Integral III	Probabilidad y Estadística I	Estructuras de Datos en Java
Álgebra Lineal II	Probabilidad I	Álgebra Superior II	Lenguajes de Programación	Aplicaciones a la Teoría del Interés	Cálculo II
Probabilidad I	Teoría del Seguro	Ideas e Instituciones Políticas y Sociales III	-Probabilidad I	Computación III	Algebra Lineal
Seguro de Daños	Programación II	Problemas de la Civilización Contemporánea II	Seguro de Daños	Seguros II	Estudio General II (C. SOCIALES)
		Administración I		Programación Lineal	Segundo Idioma III

Tabla 1.2.
Planes de Estudio de la carrera de Actuaría
Asignaturas Obligatorias

ACATLAN	CIENCIAS	ITAM	UAG	UAS	UDLA
CUARTO SEMESTRE					
Cálculo Diferencial e Integral IV	Cálculo Diferencial e Integral IV	Calculo Diferencial e Integral III	Cálculo Actuarial I	Calculo Diferencial e Integral III	Optimización en Empresa e Industria I
Probabilidad II	Ecuaciones Diferenciales I	Álgebra Lineal	Cálculo Diferencial e Integral IV	Probabilidad y Estadística II	Probabilidad I
Teoría de Sistemas	Probabilidad II	Cálculo de Probabilidades I	Ecuaciones Diferenciales I	Sistemas Financieros	Ecuaciones Diferenciales Ordinarias
Métodos Numéricos	Matemáticas Actuariales del Seguro de Personas I	Historia Socio-Política de México	-Estadística I	Ecuaciones Diferenciales	Cálculo III
Cálculo Actuarial I	Finanzas I	Matemáticas Financieras I	Estructura de Datos	Microeconomía	Macroeconomía I
		Seguros de Personas	Filosofía de la Ciencia	Modelos Actuariales de Vida I	Estudios Generales II (NEGOCIOS)
QUINTO SEMESTRE					
Estadística I	Análisis Matemático I	Sistemas Dinámicos I	Análisis Matemático	Probabilidad y Estadística III	Optimización en Empresa e Industria II
Ecuaciones Diferenciales	Investigación de Operaciones	Cálculo Actuarial I	Antropología Filosófica	Teoría de la Inversión	Probabilidad II
Investigación de Operaciones	Estadística I	Cálculo de Probabilidades II	Bases de Datos	Macroeconomía	Matemáticas Actuariales I
Cálculo Actuarial II	Matemáticas Actuariales del Seguro de Personas II	Problemas de la Realidad Mexicana Contemporánea	Cálculo Actuarial II	Modelos Actuariales de Vida II	Calculo IV
Contabilidad General	Finanzas II	Algorítmica y Programación	-Estadística II	Teoría de Decisiones	Contabilidad General
Sociedad y Política del México Actual		Matemáticas Financieras II	Microeconomía	Investigación de Operaciones	Estudios Generales IV (CIENCIAS)
				Algoritmos Numéricos	
SEXTO SEMESTRE					
Finanzas I	Optativa	Cálculo Actuarial II	Cálculo Actuarial III	Finanzas Corporativas	Optimización Avanzada
Demografía	Optativa	Cálculo Numérico I	Finanzas I	Métodos Estadísticos I	Procesos Estocásticos
Estadística II	Estadística II	Estadística Matemática	Investigación de Operaciones I	Demografía	Matemáticas Actuariales II
Economía Matemática I	Matemáticas Actuariales del Seguro de Da nos	Procesos Estocásticos I	Macroeconomía	Matematicas de Riesgo I	Inferencia Estadística
Teoría del Riesgo	Economía I	Legislación de Seguros	Probabilidad II	Optativa	Administración de Carteras de Inversión
Administración General		Seguro de Daños	Programación de Sistemas	Optativa	Análisis Matemático I

Tabla 1.2.
Planes de Estudio de la carrera de Actuaría
Asignaturas Obligatorias

ACATLAN	CIENCIAS	ITAM	UAG	UAS	UDLA
SEPTIMO SEMESTRE					
Métodos y Técnicas de Investigación	Optativa	Modelos Actuariales I	Administración I	Metodos Estadísticos II	Proyecto Actuarial
Materia Obligatoria 1 Preespecialidad	Optativa	Cálculo Actuarial III	Análisis Numérico	Administración	Teoría General del Seguro
Materia Obligatoria 2 Preespecialidad	Análisis Numérico	Matemática Demográfica	Calculo Actuarial IV	Proyecto I	Matemáticas Actuariales III
Materia Optativa 1	Seguridad Social	Estadística Aplicada I	Contabilidad General	Optativa	Métodos Estadísticos I
Materia Optativa 2	Demografía I	Contabilidad de Seguros	Metodología de la Investigación	Optativa	Demografía
Materia Optativa 3		Optativa	-Optativa I	Optativa	Optativa I
					Optativa II
					Servicio Social
OCTAVO SEMESTRE					
Seminario de Investigación Actuarial	Optativa	Planes de Beneficios	Aspectos Socioeconómicos de México	Proyecto II	Tesis I
Materia Obligatoria 3 Preespecialidad	Optativa	Estadística Aplicada a la Actuaria	Ética Profesional	Optativa	Pensiones
Materia Obligatoria 4 Preespecialidad	Teoría del Riesgo	Estadística Aplicada II	Muestreo	Optativa	Teoría del Riesgo
Materia Optativa 4	Pensiones Privadas	Optativa	-Optativa II	Optativa	Métodos Estadísticos II
Materia Optativa 5	Administración	Optativa	-Optativa III		Optativa III
Materia Optativa 6		Optativa	-Optativa IV		Simulación
					Servicio Social
NOVENO SEMESTRE					
					Tesis II
					Estudio General V (HUMANIDADES)
					Servicio Social

Tabla 1.2.
Planes de Estudio de la carrera de Actuaría
Asignaturas por Especialidad

ACATLAN	CIENCIAS	ITAM	UAG	UAS	UDLA
ESTADÍSTICA Y ECONOMÍA					
Procesos Estocásticos I	Muestreo				Microeconomía I
Economía Matemática II	Análisis Multivariado				Microeconomía II
Análisis de Regresión	Análisis de Regresión				Macroeconomía II
Modelos Microeconómicos	Diseño de Experimentos				Análisis y Evaluación de Proyectos
					Economía de Mercados Financieros
					Finanzas Corporativas
					Mercado de Valores
					Valuación Financiera
FINANZAS					
Finanzas Públicas I	Procesos Estocásticos I				Administración Financiera I
Finanzas II	Productos Financieros Derivados				Administración Financiera II
Finanzas Públicas II	Finanzas Corporativas				Proyectos de Financiamiento e Inversiones
Planeación Financiera	Carteras de Inversión				
SEGUROS					
Principios y Prácticas de Mercadotecnia	Legislación en el Seguro Privado y Social				
Organización y Programación Administrativa	Reaseguro				
Legislación de Seguros	Administración de Riesgos				
Cálculo Actuarial de Modelos Dinámicos	Seminario de Matemáticas Actuariales Aplicadas				
COMPUTACIÓN					
	Ingeniería de Software				Lenguajes Formales y Autómatas
	Bases de Datos				Prog. Funcional Lógica
	Lenguajes de Programación				
	Inteligencia Artificial				
	Redes de Computadoras				

Tabla 1.2.
Planes de Estudio de la carrera de Actuaría
Asignaturas por Especialidad

ACATLAN	CIENCIAS	ITAM	UAG	UAS	UDLA
INVESTIGACION DE OPERACIONES Y PLANEACION					
	Análisis de Redes				
	Programacion Lineal				
	Programación Entera				
	Planeación Estrategica				
ACTUARIA					
					Seminario de Estadística
					Teoria de la Credibilidad
					Seminario de Optimización
INGENIERIA INDUSTRIAL					
					Control de Calidad
					Diseño de Experimentos
					Sistemas de Logistica
					Sistemas de Informacion
					Sistemas de Producción I
					Sistemas de Producción II
					Sistemas de Producción III
MATEMATICAS					
					Matemáticas Discretas II
					Análisis Matemático II
					Variable Compleja
					Calculo Estocástico
					Calculo Científico
					Métodos Numéricos
NEGOCIOS					
					Fundamentos de Mercadotecnia
					Análisis de Mercados II
					Análisis de Mercados III
					Marketing Estratégico
					Admón. de la Información I
					Admón. de la Información II
					Dirección de Sistemas de Trabajo
					Marketing de Servicios
					Administración de Proyectos
					Administración del Conocimiento

Tabla 1.2.
Planes de Estudio de la carrera de Actuaría
Optativas

ACATLAN	CIENCIAS	ITAM	UAG	UAS	UDLA
OPTATIVAS					
Administración del Riesgo	Auditoria Actuarial		Teoria de Gráficas y Redes	Administración de Recursos Humanos	
Analisis de Estados Financieros	Finanzas		Administración II	Administración de Riesgos	
Analisis Econometrico	Mercadotecnia de Seguros		Administración de Riesgos	Algebra Lineal II	
Aplicaciones a las Matemáticas Financieras	Control Estadistico de Calidad		Análisis de Estados Financieros	Analisis de Redes	
Auditoria Actuarial	Estadistica Bayesiana		Análisis de Regresion	Analisis de Regresión	
Contabilidad de Costos	Programación Dinámica		Análisis de Sistemas I	Análisis de Series de Tiempo	
Contabilidad de Seguros	Programación No Lineal		Análisis de Sistemas II	Analisis Matematico	
Diseño de Experimentos	Simulación y Control		Demografia	Contabilidad de Seguros	
Estadistica Bayesiana	Teoria de Decisiones		Econometria	Demografia Matematica	
Evaluacion de Proyectos	Teoria de Graficas		Finanzas II	Econometria	
Estadistica de Seguros	Teoria de Inventarios Reemplazo y Mantenimiento		Inteligencia Artificial	Economia Matemática	
Finanzas Internacionales	Administración Financiera		Investigación de Operaciones II	Estadistica Multivariada	
Modelos Macroeconométricos	Administración de Riesgos Financieros		Legislación de Seguros	Estructura de Datos	
Modelos y Simulación	Productos Financieros Derivados II		Pensiones I	Finanzas Internacionales	
Pensiones	Valuación de Opciones		Pensiones II	Finanzas Matemáticas	
Presupuesto de Capital	Econometria II		Planeación Financiera	Ingenieria Financiera	
Principios y Prácticas de Mercadotecnia	Temas Selectos de Economia		Programación Lineal y no Lineal	Inteligencia Artificial	
Procesos Estadisticos II	Temas Selectos de Economia II		Seminario de Estadistica	Legislación de Seguros	
Programación Dinámica	Teoria de Juegos en Economia		Teoria de Decisiones y Juegos	Matemáticas de Riesgo II	
Programación Entera	Temas Selectos de Análisis Numerico			Modelos Actuariales de Daños	
Programación Matemática				Modelos de Sobrevivencia	
Programacion no Lineal				Muestreo	
Pronósticos de Negocios				Pensiones	
Seguro de Personas				Procesos Estocásticos	
Teoria de Gráficas y Análisis de Redes				Programación no Lineal	
Teoria de Inventarios				Simulación	
				Teoria de Colas	
				Teoria de Graduación	
				Teoria de Inventarios	
				Teoria de Juegos	

Capítulo 2
Introducción a "C"

El propósito de éste capítulo es poder dar a conocer de una manera breve y concisa el origen y características del lenguaje C. Para los lectores que ya están familiarizados con algún tipo de lenguaje podrán comprender mejor algunos conceptos aquí mencionados y para todos aquellos que C representa su entrada al mundo de la programación, deben de ser pacientes, leer con cuidado y no desesperarse ya que en el transcurso de esta tesis se irán aclarando cada uno de estos conceptos; pero no olviden que para lograr esto hay que ser constantes.

2.1. Marco histórico

El lenguaje C fue inventado e implementado por primera vez por Dennis Ritchie en un DEC PDP-11 usando UNIX como sistema operativo. C es el resultado de un proceso de desarrollo comenzado con un lenguaje anterior denominado BCPL. BCPL fue desarrollado por Martín Richard que a su vez influenció otro lenguaje denominado B, inventado por Ken Thompson. B, que en los años setenta, llevó al desarrollo del C.

Durante muchos años, el estándar de C fue realmente la versión proporcionada con la versión V del sistema operativo UNIX[4]. En el verano de 1983 se estableció un comité para crear el estándar ANSI (Instituto Nacional de Estándares Americano) que definiría el lenguaje C. El proceso de normalización duró seis años (mucho más de lo que razonablemente se podía esperar).

Finalmente, el estándar ANSI fue adoptado en diciembre de 1989, comenzando a estar disponibles las primeras copias a principios de 1990. El estándar también fue adoptado por la ISO (Organización Internacional de Estándares), siendo habitual que el estándar se refiera como estándar ANSI/ISO. En 1995 se adoptó la Enmienda 1 del estándar de C, que, entre otras cosas, incorpora varias funciones de biblioteca nuevas. El estándar 1989 de C, junto con la Enmienda 1, se convirtió en el documento base del estándar de C++, definiendo el subconjunto de C presente en C++. La versión de C definida en el estándar de 1989 se refiere a menudo como C89.

Durante los años noventa, el desarrollo del estándar de C++ acaparó casi toda la atención de los programadores. Sin embargo, el trabajo sobre C continuó, siendo

[4]Descrito en el libro El lenguaje de programación C, de Brian Kernigham y Dennis Ritchie (Englewood Cliffs, N.J.: Prentice Hall, 1991)

desarrollado un nuevo estándar. El resultado final fue el estándar de 1999 para C, normalmente referido como C99. En general, C99 mantiene prácticamente todas las características de C89. Así, ¡C todavía es C! El comité de normalización se centró en dos áreas principales: la incorporación de varias bibliotecas numéricas y el desarrollo de algunas características nuevas, de uso especial pero muy innovadoras, tales como los arreglos de tamaño variable y el cualificador restrict para apuntadores . Estas innovaciones han vuelto a poner a C en primera línea del desarrollo de lenguajes de computadora.

2.2. Características del lenguaje

2.2.1. Es un lenguaje de nivel medio

A menudo se denomina al lenguaje C como lenguaje de computadora de *nivel medio*. Esto no significa que sea menos potente, más difícil de usar o menos evolucionado que lenguajes de alto nivel como Pascal. Más bien, C se clasifica como lenguaje de nivel medio porque combina elementos de lenguajes de alto nivel con el control y la flexibilidad del lenguaje ensamblador. La **Tabla 2.1** muestra donde encaja el lenguaje C en el espectro de los lenguajes.

Tabla 2.1.

Alto nivel	Ada
	Modula-2
	Pascal
	COBOL
	FORTRAN
Nivel medio	Java
	C++
	C
	FORTH
	Macro ensamblador
Bajo nivel	Ensamblador

2.2.2. Es estructurado

Si se tiene experiencia en programación, seguramente se ha oído el término *estructurado en bloques* aplicado a un lenguaje de computadora. Aunque en un sentido estricto el término lenguaje estructurado en bloques no se puede aplicar al lenguaje C[5], normalmente este lenguaje se define como sencillamente *estructurado*. Tiene similitudes estructurales con otros lenguajes estructurados como ALGOL, Pascal y Modula-2.

5 La razón por la que C no es técnicamente un lenguaje estructurado en bloques es que un lenguaje estructurado en bloques permite declarar procedimientos o funciones dentro de otros procedimientos o funciones. Dado que C no permite la creación de funciones dentro de funciones, no puede ser formalmente denominado lenguaje estructurado en bloques.

En la **Tabla 2.2** se muestran varios ejemplos de lenguajes estructurados y no estructurados:

Tabla 2.2

No estructurados	Estructurados
FORTRAN	Pascal
COBOL	C++
Prolog	C
Lisp	Java
	Modula-2
	Visual Basic

Hoy en día podemos notar que son más conocidos y más usados, los lenguajes estructurados; aunque no se haya programado en ellos estoy casi seguro que por lo menos has escuchado hablar de estos.

2.2.3. Tiene un excelente Compilador

C es un compilador (en el apartado 2.3 haré mención de lo que es esto) pequeño, eficiente y tiene un reducido número de palabras clave (también llamadas reservadas) las cuales constituyen las órdenes que conforman el lenguaje C. Por ejemplo, C89[6] define 32 palabras clave y C99[7] tan sólo añade cinco más. Los lenguajes de alto nivel suelen tener muchas palabras clave. Como comparación, basta considerar que la mayoría de las versiones del lenguaje BASIC tienen más de 100 palabras clave.

2.2.4. Es portable

El código de C es muy portable. La *portabilidad* significa que es posible adaptar el software escrito para un tipo de computadora o sistema operativo en otro. Por ejemplo, si un programa escrito para DOS se puede llevar fácilmente a Windows 2000, entonces el programa es portable.

Esta es una de las características más apreciadas de C, gracias a que deja en manos de librerías las funciones dependientes de la máquina, ¡y todo ello sin restringir el acceso a dicha máquina!

6 Versión de C en el año 1989.
7 Versión de C en el año 1999, una versión que comprueba que C sigue evolucionando y que seguirá siendo útil por lo menos por unos años mas.

2.2.5. Utiliza bibliotecas estándar

El lenguaje C es muy simple. Carece de tipos y servicios que forman parte de otros lenguajes. No tiene tipo booleano, ni manejo de cadenas, ni manejo de memoria dinámica.

No obstante, el estándar de C define un conjunto de bibliotecas de funciones, que necesariamente vienen con todo el entorno de compilación de C y que satisfacen estos servicios elementales.

Las interfaces de estos servicios vienen definidas en unos ficheros cabeceras (*header files*) El nombre de estos ficheros suele terminar en **.h**

Algunos ejemplos de los servicios proporcionados por las bibliotecas estándares son:

- ✓ entrada y salida de datos (stdio.h)
- ✓ manejo de cadenas (string.h)
- ✓ memoria dinámica (stdlib.h)
- ✓ rutinas matemáticas (math.h)

2.2.6. Tiene construcciones algorítmicas

Un lenguaje estructurado permite muchas posibilidades en programación. Implementa directamente varias construcciones de ciclos, tales como while(), do-while() y for(). En un lenguaje estructurado, el uso de go to o bien está prohibido o se desaprueba, no siendo la forma normal de control (en la forma en que lo es en BASIC y FORTRAN, por ejemplo) Un lenguaje estructurado permite colocar las instrucciones en cualquier parte de una línea y no requiere un estricto concepto de campos (como en algunos FORTRAN antiguos).

El componente estructural principal de C es la función; una subrutina independiente. En C, las funciones son los bloques constitutivos en los que se desarrolla toda la actividad de los programas. Permiten definir las tareas de un programa y codificarlas por separado, haciendo que los programas sean modulares. Una vez que se ha creado una función que trabaja perfectamente, se puede aprovechar en distintas situaciones, sin crear efectos secundarios en otras partes del programa. El hecho de poder crear funciones independientes es algo extremadamente crítico en grandes proyectos donde es importante que el código de un programador no afecte accidentalmente al de otro.

Otra forma de estructuración y agrupación de código en C viene dada por el uso de bloques de código. Un *bloque de código* es un grupo de instrucciones de un programa conectadas de forma lógica que es tratado como una unidad. En C se

crean bloques de código colocando una serie de instrucciones entre llaves. En este ejemplo:

```
if (x < 10)
{
    printf("Demasiado bajo, pruebe de nuevo.\n");
    scanf("%d", &x);
}
```

las dos instrucciones después del if y entre las llaves se ejecutan juntas si x es menor que 10.

Estas dos instrucciones junto con las llaves representan un bloque de código. Se trata de una unidad lógica: no se puede ejecutar una de las instrucciones sin la otra. Los bloques de código permiten implementar con claridad, elegancia y eficiencia muchos algoritmos. Más aún, ayudan al programador a asimilar el concepto de la verdadera naturaleza del algoritmo.

C también contiene estructuras de decisión como de falso y verdadero como es el caso del *if()* y de decisión múltiple como switch().

2.2.7. Contiene estructuras de datos

Todos los lenguajes de alto nivel contemplan el concepto de tipos de datos. Un tipo de dato define un conjunto de valores que puede tener una variable (este concepto es igual que en las matemáticas) junto con un conjunto de operaciones que se pueden realizar sobre esa variable. Algunos tipos de datos comunes son los enteros, los caracteres (letras, números, símbolos, etc.) y los reales. Aunque el lenguaje C tiene varios tipos de datos básicos incorporados, no se trata de un lenguaje fuertemente tipificado, como los son Pascal y Ada. C permite casi cualquier conversión de tipos.

A diferencia de la mayoría de los lenguajes de alto nivel, C no lleva a cabo una comprobación de errores en tiempo de ejecución. Por ejemplo, no comprueba que no se sobrepasen los límites de los arreglos (en matemáticas conocidos como matrices). Es el programador el único responsable de llevar a cabo esas comprobaciones.

2.2.8. Acepta comentarios

En el C original, los comentarios tienen la forma */* cualquier texto */* , se pueden extender varias líneas y no se pueden anidar entre sí.

Ejemplos:

```
{
        /*        Esto es un comentario
                  que ocupa varias líneas
        */
        /*        Comentario de una línea     */
}
```

2.2.9. Es útil en varios campos

Estas y otras características lo hacen adecuado para la programación en áreas tales como:

- ✓ programación de sistemas
- ✓ estructuras de datos y sistemas de bases de datos
- ✓ aplicaciones científicas
- ✓ software gráfico
- ✓ análisis numérico

2.2.10. Es un lenguaje para programadores

Sorprendentemente, no todos los lenguajes son para los programadores. Considérense los ejemplos clásicos de lenguajes para no programadores: COBOL y BASIC. COBOL no fue diseñado para mejorar las habilidades de los programadores, ni para mejorar la fiabilidad del código producido, ni siquiera para aumentar la velocidad con la que se pudiera producir código. Más bien, COBOL se diseñó para que los no programadores pudieran leer y comprender el programa. BASIC fue creado esencialmente para permitir a los no programadores programar una computadora para resolver problemas relativamente sencillos.

Por contra, C fue creado, influenciado y probado en vivo por programadores profesionales. El resultado final es que C da al programador lo que el programador quiere: pocas restricciones, pocas reglas, estructuras de bloques, funciones independientes y un compacto conjunto de palabras clave. Usando C, un programador puede casi alcanzar la eficiencia del código ensamblador junto con la estructuración de Pascal o Modula-2.

El hecho de que el lenguaje C sea usado a menudo en vez de ensamblador es un factor determinante de su popularidad entre los programadores.

Inicialmente, C fue usado para la programación de sistemas. Un programa del sistema es una parte del sistema operativo de la computadora o de sus utilidades de soporte, tales como los editores, los compiladores, los enlazadores y similares. A medida que el lenguaje C creció en popularidad, muchos programadores comenzaron a usarlo para programar cualquier tipo de tarea debido a su portabilidad y eficiencia; y porque les gustaba. En el momento en el que fue creado, C se convirtió en un gran paso adelante en el mundo de los lenguajes de programación. En todos estos años que han transcurrido, C ha demostrado que está preparado para cualquier tarea.

Con la aparición de C++, algunos programadores pensaron que C dejaría de existir como lenguaje con entidad propia. No ha sido así. En primer lugar, no todos los programadores necesitan aplicar los mecanismos de orientación a objetos de C++. Por ejemplo, las aplicaciones típicas de sistemas empotradas todavía se suelen desarrollar con C. En segundo lugar, muchos de los programas existentes en todo el mundo todavía funcionan a partir de código C, requiriendo ser mantenidos y mejorados. En tercer lugar, como prueba el nuevo estándar C99, C todavía es un campo en el que se producen innovaciones de primera línea. Aunque sin duda C será recordado siempre como la base de C++, también se le reconocerá por sí mismo como uno de los lenguajes más importantes.
Para los estudiantes que se inician en el estudio de C, la visión del lenguaje les causa miedo, C tiene fama de ser un lenguaje difícil, nada más lejos de la verdad. Una vez que se entiende cómo trabaja, es fácil dominarlo, por algo C es el preferido de muchos programadores que lo consideran un lenguaje "elegante". Por no ser un lenguaje de alto nivel, C le relega al programador cierto grado de responsabilidad en el desarrollo de los programas.

2.3. Compiladores frente a Intérpretes

Es importante comprender que un lenguaje de computadora define la naturaleza de un programa y no la forma en que el programa se ejecuta. Dos son los métodos generales de ejecución de un programa. Puede ser *compilado o interpretado*. Aunque los programas escritos en cualquier lenguaje de programación pueden ser compilados o interpretados, algunos lenguajes han sido diseñados más para una forma de ejecución que para la otra. Por ejemplo, Java fue diseñado pare ser interpretado y C para ser compilado. Sin embargo, en el caso de C es importante tener en cuenta que ha sido especialmente optimizado como lenguaje compilado. Aunque se han escrito algunos intérpretes de C para ciertos entornos. C fue desarrollado con la compilación en mente. Por esto, es casi seguro que lo que se utiliza para desarrollar programas en C es un compilador de C y no un intérprete. Como puede que no todos los lectores tengan clara la diferencia entre un compilador y un intérprete, a continuación se proporciona una descripción para intentar aclarar el asunto.

En su forma más sencilla, un intérprete lee el código fuente de un programa línea a línea, y ejecuta las instrucciones específicas contenidas en esa línea. Ésta es la

forma en la que trabajaban las primeras versiones de BASIC. En lenguajes como Java, el código fuente de un programa se convierte primero en una forma intermedia que es la que se interpreta entonces. En cualquier caso se requiere la presencia de un intérprete de tiempo de ejecución para que se pueda ejecutar el programa.

Un compilador lee el programa entero y lo convierte a código objeto, que es una traducción del código fuente del programa a una forma que puede ser ejecutada directamente por la computadora. El código objeto también se suele denominar código binario o código máquina. Una vez que el programa está compilado, las líneas de código fuente dejan de tener sentido durante la ejecución del programa.

En general, un programa interpretado se ejecuta más despacio que un programa compilado. Recuérdese que un compilador convierte el código fuente de un programa en código objeto que se puede ejecutar directamente en la computadora. Por tanto, la compilación sólo pasa factura una vez, mientras que el código interpretado tiene un precio cada vez que se ejecuta un programa.

2.3.1. Forma de un programa en C

Todos los programas en C consisten en una o más funciones (las funciones son bloques de C donde se produce toda la actividad del programa). Como regla general, la función que forzosamente debe estar presente es la denominada main(), siendo la primera función que es invocada cuando comienza la ejecución del programa. En código C bien escrito, main() contiene lo que en esencia el programa hace. Ese esbozo está compuesto por llamadas a funciones. Aunque main() no es una palabra clave, se ha de tratar como si lo fuera. Por ejemplo, no se debe intentar utilizar main() como nombre para una variable, ya que probablemente confundirá al compilador.

La **tabla 2.3** lista las 32 palabras clave definidas por el estándar C89. Éstas son también las palabras clave que constituyen el subconjunto de C presente en C++.

Tabla 2.3

Palabras clave definidas en C89

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	go to	size of	volatile
do	if	static	while

y la **tabla 2.4** muestra las palabras clave añadidas por C99.

Tabla 2.4

Palabras clave añadidas por C99.

Bool	Imaginary
restrict	Complex
inline	

Obs. Las palabras clave, combinadas con la sintaxis formal de C, conforman el lenguaje de programación C.

En C, las mayúsculas y las minúsculas son diferentes; por ejemplo else es una palabra clave; ELSE no lo es. Una palabra clave no debe ser usada para otro propósito en un programa en C; es decir, no puede servir como nombre de variable o de función.

2.3.2. La biblioteca y el enlace

Técnicamente hablando, se puede crear un programa en C que sea funcional y útil y que consista únicamente en instrucciones que se construyan con las palabras clave de C. Sin embargo, esto es bastante raro ya que C no proporciona palabras clave para llevar a cabo cosas como operaciones de entrada/salida (E/S). Por ello, la mayoría de los programas incluyen llamadas a varias funciones contenidas en la biblioteca estándar de C.

Todos los compiladores de C incorporan una biblioteca estándar que proporciona las funciones que se necesitan para llevar a cabo las tareas más usuales. El estándar de C especifica un conjunto mínimo de funciones que deben ser proporcionadas por todos los compiladores. Sin embargo, es muy probable que el compilador contenga muchas otras funciones. Por ejemplo, la biblioteca estándar no define ninguna función de gráficos, pero seguramente el compilador que se tenga incluirá algunas.

Cuando se llama a una función de la biblioteca, el compilador de C "recuerda" su nombre. Más tarde, el enlazador combina el código que se ha escrito con el código objeto que ya se encuentra en la biblioteca estándar. Este proceso se denomina enlace. Algunos compiladores tienen su propio enlazador, mientras que otros usan el ensamblador estándar proporcionado por el sistema operativo.

Las funciones de la biblioteca se encuentran en formato reubicable. Esto significa que las direcciones de memoria de las diferentes instrucciones de código máquina no han sido definidas de forma absoluta; sólo se mantiene información sobre

desplazamientos. Cuando se enlaza un programa con funciones de la biblioteca estándar, se usan esos desplazamientos de memoria para obtener las direcciones reales. Existen varios manuales y libros técnicos que explican más detalladamente este proceso. Sin embargo, no se necesita una mayor explicación sobre el proceso concreto de reubicación para poder programar en C.

Muchas de las funciones necesarias para escribir programas se encuentran en la biblioteca estándar. Se comportan como bloques constitutivos que simplemente hay que combinar. Si se escribe una función que se va a usar repetidas veces, también puede ser situada en la biblioteca.

2.3.4. Compilación separada

La mayoría de los programas cortos de C están enteramente contenidos en un archivo fuente. Sin embargo, a medida que aumenta la longitud del programa, también lo hace el tiempo de compilación (y los altos tiempos de compilación no están hechos para los programadores impacientes). Por ello, C permite partir un programa en muchos archivos y que cada uno sea compilado por separado. Una vez que han sido compilados todos los archivos, se enlazan entre sí, junto con las rutinas de la biblioteca, para formar el código objeto completo. La ventaja de la compilación separada es que un cambio en el código de uno de los archivos no requiere la recopilación del programa entero. En cualquier proyecto, excepto en los más simples, el ahorro de tiempo es sustancial. La compilación separada, además, permite que en los proyectos trabajen fácilmente varios programadores en equipo, como también se constituye en un medio de organización del código en los grandes proyectos.

2.3.5. Compilación de un programa en C

La creación de una forma ejecutable de un programa en C consiste en estos tres pasos:

1. Creación del programa.
2. Compilación del programa.
3. Enlace del programa con las funciones que se necesiten de la biblioteca.

Actualmente, la mayoría de los compiladores de C proporcionan entornos integrados de desarrollo que incluyen un editor. La mayoría también incluyen compiladores independientes. Para las versiones independientes se debe disponer de un editor aparte para crear el programa. En cualquier caso, hay que tener cuidado: los compiladores sólo admiten como entrada archivos de texto estándar. Por ejemplo, el compilador no aceptará archivos creados con ciertos procesadores de texto, debido a que contendrán códigos de control y caracteres no imprimibles.

El método exacto que se utilice para compilar un programa dependerá del compilador de C que se uso. Además, la forma en que se lleve a cabo el enlace variará entre distintos compiladores y entornos. Por ejemplo, el enlazador puede estar incluido como parte del compilador o puede ser una aplicación independiente. En el manual de usuario se proporcionarán todos estos detalles.

Capítulo 3
Conceptos Básicos

En este Capítulo se comenzará a entrar en materia de programación, a formalizar algunos de los conceptos básicos y aprenderemos a usarlos adecuadamente.

3.1 . Identificadores

En C, los nombres de las variables, funciones, etiquetas y otros objetos definidos por el usuario se denominan identificadores. La longitud de un identificador puede variar entre uno y varios caracteres. El primer carácter debe ser una letra o un símbolo de subrayado y los caracteres siguientes pueden ser letras, números o símbolos de subrayado;.ejemplos:

Suma
Total
Pma_Dif
_12345

En C, los identificadores pueden ser de cualquier longitud. Sin embargo, no necesariamente todos los caracteres han de ser significativos (caracteres que reconoce C). C define dos clases de identificadores: externos e internos. En C89, al menos los seis primeros caracteres de los identificadores externos y los primeros 31 de los identificadores internos han de ser significativos. C99 ha aumentado esas cifras. En C99, un identificador externo tiene al menos 31 caracteres significativos y uno interno al menos 63. En los identificadores las minúsculas y las mayúsculas se tratan como distintas. Así, casa, Casa y CASA son tres identificadores distintos.

Un identificador no puede coincidir con una palabra clave de C y no debe ser el mismo nombre que el de alguna función de la biblioteca de C.

3.2. Tipos de datos básicos

C89 define cinco tipos de datos básicos que son:

Tabla 3.1

Tipo	Como se Declara	Descripción
Carácter	char	Todo tipo de letra, numero o símbolo.
Entero	int	Números enteros positivos y negativos
Real	float	Números reales
Real de Doble precisión	double	Números reales con doble precisión
Sin valor	void	Conjunto vacío

Todos los demás tipos de datos se basan en alguno de esos tipos

Nota: A los cinco tipos de datos básicos definidos por C89, C99 añade tres más: **_Bool, _Complex e _Imaginary.**

3.2.1 Modificadores

A excepción del tipo void, los tipos de datos básicos pueden tener distintos modificadores precediéndolos. Un modificador se usa para alterar el significado del tipo base de forma que se ajuste más precisamente a las necesidades de cada momento. En la **Tabla 3.2** se muestra la lista de modificadores:

Tabla 3.2

Modificador
signed
unsigned
long
Short

La **Tabla 3.3** muestra todas las combinaciones de tipos que se ajustan al estándar de C, junto con sus intervalos mínimos y tamaños típicos en bits. Téngase en cuenta que la tabla muestra los intervalos mínimos establecidos para los tipos, no sus intervalos típicos. Por ejemplo, en las computadoras que utilizan la aritmética de complemento a dos (casi todas), un entero tendrá por lo menos un intervalo de entre 32.767 y -32.768.

El uso de signed con enteros, aunque se permite, es redundante porque la declaración implícita de enteros asume números con signo. El uso más importante de signed es para modificar char en implementaciones en las que char no tenga signo por defecto.

Tabla 3.3

Tipo	Tamaño aproximado en bits	Intervalo mínimo (Valores Permitidos)
Char	8	[-127 , 127]
unsigned char	8	[0 , 255]
signed char	8	[-127 , 127]
Int	16 o 32	[-32,768 , 32,767]
unsigned int	16 o 32	[0, 65 , 535]
signed int	16 o 32	[-32,768 , 32,767]
Short int	16	[-32,768 , 32,767]
unsigned short int	16	[0 , 65,535]
signed short int	16	[-32,768 , 32,767]
long int	32	[-2,147,483,647 , 2,147,483,647]
long long int	64	[$-(2^{63}-1)$, $2^{63}-1$] (Añadido por C99)
signed long int	32	[-2,147,483,647 , 2,147,483,647]
unsigned long int	32	[0 , 4,294,967,295]
unsigned long long int	64	[0 , $2^{64}-1$] (Añadido por C99)
Flota	32	[1E-37 , 1E+37] con seis dígitos de precisión
Double	64	[1E-37 , 1E+37] con diez dígitos de precisión
long double	80	[1E-37 , 1E+37] con diez dígitos de precisión

Cuando se utiliza un modificador de tipo por sí solo (es decir, sin preceder a ningún tipo básico), entonces se asumen enteros por default; como se muestra en la **Tabla 3.4.**

Tabla 3.4

Especificador	Equivale a
signed	signed int
unsigned	unsigned int
long	long int
short	short int

Ahora es tiempo de realizar nuestro primer programa.

Programa 1

```
# include <stdio.h>

/* Este programa imprime en la pantalla el tamaño en bytes de los diferentes tipos de datos en C */

void main()
{
  clrscr ();
  printf("\n Tamaño (en bytes) de los diferentes tipos de datos en C (bajo MS--DOS) ");
  printf("\n Tamaño :unsigned char          = %d bytes", sizeof(unsigned char));
  printf()("\n Tamaño: char                 = %d bytes", sizeof (char));
  printf("\n Tamaño :unsigned short         = %d bytes", sizeof (unsigned short));
  printf()("\n Tamaño :short                = %d bytes", sizeof (short));
  printf("\n Tamaño :unsigned int           = %d bytes", sizeof(unsigned int));
  printf("\n Tamaño :int                    = %d bytes",sizeof(int));
  printf("\n Tamaño :unsigned long          = %d bytes",sizeof(unsigned long));
  printf("\n Tamaño :long                   = %d bytes",sizeof(long));
  printf("\n Tamaño :float                  = %d bytes",sizeof(float));
  printf("\n Tamaño :double                 = %d bytes",sizeof(double));
  printf("\n Tamaño :long double            = %d bytes",sizeof(long double));
}

/* Por definición

    a) BIT es la unidad mínima de Almacenamiento.
    b) BYTE es el conjunto de n bits, donde n depende del sistema (usualmente
       n=8).
*/
```

El código de este programa se interpreta de la siguiente manera:

- Declara una biblioteca que es el archivo que incluye algunas funciones que podrán ser usadas en el programa. Por ejemplo <stdio.h> nos proporciona el soporte para entrada y salida de archivos; esta librería es quizá una de las más importantes y usadas por lo que la colocaré por default en todos los programas subsecuentes. Observemos que la forma correcta de declarar una biblioteca es " #include <biblioteca > " donde # include debe colocarse siempre antes de declararla, y la cual deberá ir entre < >.

- Coloca un comentario que sirve de guía para la persona que lee el código del programa. Hay que recordar que debe colocarse entre " /* */ "

- Declara la función principal "void main()" que incluye todo lo que está entre las llaves (más adelante cuando se vean funciones se analizará porque se pone la palabra void al principio de la misma).

- Limpia la pantalla con clrscr();
- Con printf() despliega en la pantalla la información:

```
(Inactive C:\TCWIN\BIN\PROG1.EXE)

Tama¤o (en bytes) de los diferentes tipos de datos en C (bajo MS-DOS)
Tama¤o :unsigned char  = 1 bytes
Tama¤o :char           = 1 bytes
Tama¤o :unsigned short = 2 bytes
Tama¤o :short          = 2 bytes
Tama¤o :unsigned int   = 2 bytes
Tama¤o :int            = 2 bytes
Tama¤o :unsigned long  = 4 bytes
Tama¤o :long           = 4 bytes
Tama¤o :float          = 4 bytes
Tama¤o :double         = 8 bytes
Tama¤o :long double    = 10 bytes
```

Obs. Hay que poner mucha atención en el " ; " que se coloca después de cada instrucción ya que sin éste el programa no compilará y en consecuencia no correrá. Se puede hacer la prueba quitándole el " ; " a una línea. La función sizeof (tipo de dato) despliega el tamaño de este.

En el siguiente apartado se verá cómo funciona printf() .

3.3. printf()

La función printf() escribe datos en la consola (salida estándar) y realiza la entrada con formato; ésto es, puede leer y controlar los datos en varios formatos. Además de operar sobre cualquiera de estos tipos de datos existentes, incluyendo cadenas de caracteres terminadas en nulo.

La estructura general de printf()es la siguiente:

```
printf( "cadena de formato", arg1, arg2, ... ,argn);
```

La cadena de formato está formada por dos tipos de elementos. El primer tipo está compuesto por los caracteres que se mostrarán en la pantalla. El segundo tipo contiene los argumentos que corresponden a los especificadores colocados dentro del primero.

Un especificador de formato empieza con un signo de porcentaje y sigue con el código de formato. Debe haber exactamente el mismo número de argumentos adicionales que de especificadores de formato, y éstos se hacen corresponder con los argumentos en orden de aparición de izquierda a derecha. Por ejemplo, esta llamada a printf()

```
printf("La suma de  %d  +  %d  es igual a  %d", 3, 5, 8);
```

muestra:

```
La suma de 3 + 5 es igual a  8
```

donde tenemos 3 especificadores de formato y 3 argumentos los cuales se corresponden en el orden mencionado anteriormente.

Obs. Una línea como la mostrada anteriormente debe de ir dentro de un programa, porque por sí sola no sirve para nada.

3.3.1. Especificadores de formato de printf()

Estos especificadores nos permiten indicar que tipo de dato se va a imprimir; como en el ejemplo anterior 3, 5 y 8 son enteros, ocupamos %d, pero si hubieran sido 3.5, 6.9 y 10.4 hubiéramos utilizado %f.

Como veremos mas adelante cada uno de los especificadores que se muestran en la Tabla 3.5 nos permiten desplegar algún tipo de dato especifico.

Tabla 3.5

Código	Formato
%a	Salida hexadecimal en la forma Oxh.hhhhp+d (sólo en C99).
%A	Salida hexadecimal en la forma OXh.hhhhP+d (sólo en C99).
%c	Carácter.
%d	Enteros decimales con signo.
% i	Enteros decimales con signo.
%e	Notación científica (e minúscula).
%E	Notación científica (E mayúscula).
%f	Real en coma flotante.
%g	Usa %e o %f, el más corto.
%G	Usa %E o %f, el más corto.
%o	Octal sin signo.
%s	Cadena de caracteres.
%u	Enteros decimales sin signo.
%x	Hexadecimales sin signo (letras minúsculas).
%X	Hexadecimales sin signo (letras mayúsculas).
%p	Muestra un apuntador.
%n	El argumento asociado debe ser un apuntador a un entero. Este especificador hace que se guarden en ese entero el número de caracteres escritos (hasta el momento en el que se encuentra el %n).
%%	Imprime el signo %.

3.3.2. Impresión de caracteres

Para imprimir un solo carácter se utiliza %c, esto hace que se muestre su correspondiente argumento en la pantalla sin modificaciones; ejemplo:

```
printf( "Esto es un carácter %c ", a);
```

En la pantalla se imprimirá **Esto es un carácter a**

Para imprimir una cadena se utiliza %s; ejemplo:

```
printf( "Cadena %s ", 'Hola');
```

En la pantalla se imprimirá **Cadena Hola**

3.3.3. Impresión de números

Se puede utilizar %d o %i para indicar un número decimal con signo. Estos especificadores de formato son equivalentes; se han mantenido ambos por razones históricas, una de las cuales es el deseo de mantener una relación de equivalencia con los especificadores de formato de scanf() (mas adelante veremos para que se utiliza este).

Para mostrar un valor sin signo se utiliza %u.

El especificador de formato %f imprime números reales en coma flotante. El argumento correspondiente ha de ser de tipo "double".

Los especificadores %e y %E indican a printf() que muestre su argumento "double" en notación científica. Los números representados en notación científica tienen la siguiente forma general:

x.dddddE+/- zz

donde "x" es la constante, "E" la base 10 y "zz" el exponente.

Si se quiere imprimir la letra "E" en mayúscula, se utiliza el formato %E; en caso contrario se utiliza %e.

Se le puede indicar a la función printf() que utilice %f o %e mediante los especificadores de formato %g o %G. Esto hace que printf() seleccione el especificador de formato que genere la salida más corta. Cuando proceda, se usa %G si se quiere imprimir la "E" en mayúscula; en caso contrario se utiliza %g. Por ejemplo si queremos desplegar los números del 1 al 10,000,000,000 con la salida mas corta (i.e. la salida que ocupe menos caracteres;) obtendremos:

1 10 100 1000 10000 100000 le+006 le+007 le+008 1e+009

Observa como los números del 1 al 10000 tienen menos de 6 caracteres, el número 100000 ya tiene 6 caracteres igual que 1e+005; en este caso se da prioridad a 100000 y todos los números mayores a 100000 tienen mas de 6 caracteres lo cual obliga a utilizar como salida mas corta a la Notación Científica.

Se pueden mostrar enteros sin signo en formato octal o hexadecimal utilizando %o y %x, respectivamente. Como el sistema de numeración hexadecimal utiliza las letras de la A a la F para representar los números del 10 al 15, se pueden imprimir estas letras tanto en mayúscula como en minúscula. Para mayúsculas se utiliza el especificador de formato %X; para minúsculas %x, como se muestra a continuación.

```
#include <stdio.h>

int main(void)
{
unsigned num ;

for (num=0; num<16; num++)
        {
        printf(%o ",num);
        printf(%x ",num);
        printf(%X\n ",num);
        }
return 0;
}
```

La salida es la siguiente:

```
0     0     0
1     1     1
2     2     2
3     3     3
4     4     4
5     5     5
6     6     6
7     7     7
10    8     8
11    9     9
12    a     A
13    b     B
14    c     C
15    d     D
16    e     E
17    f     F
```

Obs. Para este ejemplo hemos hecho uso del ciclo for el cual veremos en el capítulo 4. Por el momento lo más importante es concentrarse en la salida que produce el programa así como en el uso de los modificadores.

3.3.4. Modificadores de formato

Muchos especificadores de formato aceptan modificadores que alteran su significado ligeramente. Por ejemplo, se puede especificar la longitud mínima de un campo, el número de decimales y el ajuste a la izquierda. El modificador de formato se sitúa entre el signo de porcentaje y el código de formato. A continuación se tratan estos modificadores.

3.3.4.1. El especificador de longitud mínima de campo

Un entero situado entre el signo % y el código de formato actúa como un especificador de longitud mínima de campo[8]. Hace que se rellene la salida con espacios para asegurar que el campo alcanza una cierta longitud mínima. Si la cadena o el número es más largo que ese mínimo, se mostrará completamente.

Los huecos se rellenan con espacios por default. Si se quiere rellenar con ceros, se ha de poner un 0 antes del especificador de longitud de campo. Por ejemplo, %05d rellenará con ceros y %5d deja blancos, los números con menos de 5 dígitos para que su longitud total sea cinco.

El modificador de longitud mínima de campo se utiliza habitualmente para crear tablas en las que las columnas aparezcan alineadas. Por ejemplo, el siguiente programa genera la tabla de los cuadrados y los cubos de los números entre 1 y 10.

```
#include <stdio.h>
{
i
int i;
for(i =1; i < 11; i++)
    printf("%8d %8d %8d\n", i, i *i, i *i*I);

return 0;
}
```

A continuación se muestra la salida:

[8] Por definición se dice que un conjunto de números forman un campo de números si la suma, diferencia, producto y cociente (excluyendo la división entre cero) de dos números cualesquiera del conjunto (sean iguales o diferentes), son también elementos del mismo conjunto.

```
1    1    1
2    4    8
3    9    27
4    16   64
5    25   125
6    36   216
7    49   343
8    64   512
9    81   729
10   100  1000
```

Obs. Para este ejemplo hemos hecho uso del ciclo for() el cual veremos en el capítulo 4. Por el momento lo mas importante es concentrarse en la salida que produce el programa así como en el uso de los especificadores.

3.3.4.2. El especificador de precisión

El especificador de precisión sigue al especificador de longitud mínima de campo (sí existe). Consiste en un punto seguido de un entero. Su significado exacto depende del tipo de dato al que se aplique.

Cuando se aplica el especificador de precisión a datos en coma flotante junto con los especificadores %f, %e o %E, aquél determina el número de posiciones decimales a mostrar. Por ejemplo, %10.4f imprime un número de al menos diez caracteres con cuatro decimales. Si no se especifica precisión, se muestran implícitamente seis.

Cuando se aplica un especificador de precisión a %g o %G, especifica el número de dígitos significativos.

Si se aplica a cadenas, el especificador de precisión determina la longitud máxima del campo. Por ejemplo, %5.7s imprime una cadena con al menos cinco caracteres de longitud y no más de siete. Si la cadena es más larga que la longitud máxima del campo, se truncarán los caracteres finales.

Cuando se aplica a tipos enteros, el especificador de precisión determina el número mínimo de dígitos que aparecerán para cada número. Para alcanzar el número requerido de dígitos se añaden ceros por delante.

El siguiente programa ilustra el especificador de precisión:

```
#include <stdio.h>

int main(void)
{
printf("%.4f\n", 123.123456);
printf("%3.8d\n",1000);
printf("%10.15s\n", "Esta es una prueba sencilla.");
printf("%.2e\n", 1234.123456);
printf("%.5G\n", 123.123456);
return 0;
}
```

Produce la siguiente salida

```
123.1235
00001000
Esta es una pru
1.23e+03
123.12
```

3.3.4.3. Ajuste de la salida

Por default, todas las salidas se ajustan a la derecha. Esto es, si la longitud de campo es mayor que el dato a mostrar, el dato se coloca en la parte derecha del campo. Se puede forzar a que la salida se ajuste a la izquierda, poniendo un signo menos justamente después del porcentaje. Por ejemplo, %-10.2f hace que se ajuste a la izquierda un número en coma flotante con dos posiciones decimales en un campo de diez caracteres.

El siguiente programa ilustra el ajuste a la izquierda:

```
#include <stdio.h>

int main(void)
{
printf( " . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .\n ");
printf("ajustado a la derecha: %8d\n", 100);
printf("ajustado a la izquierda: %-8d\n", 100);

return 0;
}
```

La salida es la siguiente:

```
..............................
ajustado a la derecha      100
ajustado a la izquierda 100
```

3.3.5. Manejo de otros tipos de datos

Existen dos modificadores de formato que permiten a printf() mostrar enteros largos y cortos. Estos modificadores se pueden aplicar a los especificadores de tipo d, i, o, u y x. El modificador l indica a printf() que lo que sigue es un tipo de dato long. Por ejemplo, %ld significa que se va a mostrar un long int. El modificador h indica a printf() que se va a mostrar un entero corto (short). Por ejemplo, %hu indica que el dato es de tipo short unsigned int.

Veamos el siguiente ejemplo:

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

void main()
{
  /* Declaración de la variable largo de tipo entera larga */
  unsigned int largo = 35000;

  clrscr();

  /* Para imprimir un entero sin signo se utiliza el especificador
     de formato u */
  printf("\n\n\n%d %u", largo,largo);
  getch();
}
```

La salida es la siguiente:

```
-30536 35000
```

- Aquí podemos observar que si no declaramos la variable largo como "entera larga" podemos llegar a un resultado erróneo.

Los modificadores l y h también se pueden aplicar al especificador n, para indicar que el argumento correspondiente es un apuntador a un entero largo o corto, respectivamente.

Si se utiliza un compilador que admita las características de caracteres ampliados añadidas por la Enmienda 1 de 1995, se puede utilizar el modificador l con el formato c para indicar un carácter ampliado. También se puede utilizar el modificador l con el formato s para indicar una cadena de caracteres ampliados.

El modificador L puede preceder a los especificadores de coma flotante e, f y g, e indica que le sigue un long double.

C99 añade dos nuevos modificadores de formato: hh y ll. El modificador hh se puede aplicar a d, i, o, u, x o n. Especifica que el correspondiente argumento es un valor signed o unsigned char, o, en el caso de n, un apuntador a una variable signed char. El modificador ll también se puede aplicar a d, i, o, u, x o n. Especifica que el correspondiente argumento es un valor signed o unsigned long long int, o, en el caso de n, un apuntador a una variable long long int. C99 también permite aplicar la l a los especificadores de coma flotante, a, e, f y g, aunque no tiene efecto.

Veamos los siguientes ejemplos:

Programa 2

```
/* Este programa nos da diferentes salidas de datos mediante el uso de los especificadores
de formato */

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

void main()
{
  /* Declaración de una variable llamada distancia del tipo de dato
     entera larga con signo, inicializándola con el valor de 150000000 */

  signed long int distancia = 150000000;

  /* Para imprimir un entero largo se necesita del especificador de
     formato ld */

  printf("La distancia de la tierra <-->Sol es de %ld[Km.]\n",distancia);
  getch();
}
```

Este programa despliega en la pantalla el siguiente resultado:

```
C:\TCWIN\BIN\NONAME00.EXE
La distancia de la tierra <-->Sol es de 150000000[Km]
_
```

Para finalizar este apartado veremos el uso de getch(), getche() y getchar() por medio de el siguiente programa.

Programa 3

```
/* Este programa nos ilustra el uso de getch(), getche() y getchar() */

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

void main(void)
{
 char carácter1, carácter2, carácter3;

 clrscr();
 printf("Utilizando getch: ");
 carácter1=getch();
 printf("\nCarácter1 tiene: %c", carácter1);
 printf("\n\nUtilizando getche: ");
 carácter2=getche();
 printf("\nCarácter2 tiene: %c", carácter2);
 printf("\n\nUtilizando getchar: ");
 carácter3=getchar();
 printf("Carácter3 tiene: %c\n", carácter3);
 getch();

}
```

Este programa realiza lo siguiente:

- Me pide un carácter y cuando lo pulso no lo muestra y salta de línea automáticamente (i.e. getch() no muestra el carácter tecleado y tiene un enter por default).
- Enseguida me pide otro carácter y esta vez si me lo muestra y salta de línea automáticamente (i.e. getche() me muestra el carácter y tiene un enter integrado).
- Al final me pide otro carácter y cuando se lo doy continua el cursor en la misma línea donde puedo seguir escribiendo caracteres hasta pulsar un enter (i.e. getchar() me permite escribir y visualizar varios caracteres y no tiene un enter automático).

- Si pulsara en cada línea la Palabra Hola en los 3 casos tendría una salida de programa como la siguiente:

```
(Inactive C:\TCWIN\BIN\...)
Utilizando getch:
Caracter1 tiene: H

Utilizando getche: H
Caracter2 tiene: H

Utilizando getchar: Hola
Caracter3 tiene: H
```

3.4. scanf()

scanf() es la rutina de entrada por consola de propósito general. Puede leer todos los tipos de datos incorporados y convierte los números automáticamente al formato interno apropiado. Es como si fuera la inversa de printf().

La estructura de scanf es:

scanf (formato, &arg1, &arg2, ...);

En <formato> se especifican qué tipos de datos se quieren leer (utilizando la misma descripción de formato que en printf()), el amperson (&) debe ir antes de cualquier argn; donde las argn son las variables donde se va almacenar el dato leído y éstas se hacen corresponder con los argumentos en orden de aparición de izquierda a derecha.

Obs. Si no se anteponen los ampersons (&), el resultado puede ser desastroso. En scanf sólo van descripciones de formato, nunca texto normal. Si se quiere escribir antes un texto, hay que utilizar printf().

3.4.1. Especificadores de formato de scanf()

Los especificadores de formato de entrada van precedidos por el signo % e indican a scanf() qué tipo de dato se va a leer a continuación. Estos códigos aparecen en la **Tabla 3.6.** Los especificadores de formato se asocian en orden, de izquierda a derecha, con los argumentos de la lista de argumentos.

Tabla 3.6

Código	*Significado*
%a	Lectura de un valor en coma flotante (sólo en C99).
%c	Lectura de un único carácter.
%d	Lectura de un entero decimal.
%i	Lectura de un entero, en formato decimal, octal o hexadecimal.
%e	Lectura de un número de coma flotante.
%f	Lectura de un número de coma flotante.
%g	Lectura de un número de coma flotante.
%o	Lectura de un número octal.
%s	Lectura de una cadena.
%x	Lectura de un número hexadecimal.
%p	Lectura de un apuntador.
%n	Devolución de un valor entero igual al número de caracteres ya leídos.
%u	Lectura de un entero decimal sin signo.
%[ ]	Muestreo de un conjunto de caracteres.
%%	Lectura de un signo de porcentaje.

Obs. La función scanf() termina de leer un número cuando encuentra el primer carácter no numérico.

3.4.2. Lectura de números.

Para leer un entero se utiliza el especificador %d o %i. Para leer un número real en coma flotante representado en notación estándar o científica se utiliza %e, %f o %g. (C99 también incluye %a, que lee un número en coma flotante.)

Se puede utilizar scanf() para leer enteros en forma octal o hexadecimal utilizando las órdenes de formato %o y %x, respectivamente. Con %x puede escribirse tanto en mayúsculas como en minúsculas. De cualquier forma, se pueden introducir las letras de la A a la F para números hexadecimales. El siguiente programa lee e imprime un número octal y uno hexadecimal:

```
#include <stdio.h>
int  main (void)
{
int i, j;
scanf("%o %x", &i, &j),
printf("%o %x", i, j),
return 0;
}
```

La función scanf() termina de leer un número cuando encuentra el primer carácter no numérico.

3.4.3. Lectura de enteros sin signo

Para introducir un entero sin signo se utiliza el especificador de formato %u. Por ejemplo,

```
unsigned num;
scanf("%u", &num);
```

lee un número sin signo y coloca el valor en **num.**

3.4.4. Lectura de caracteres individuales con scanf()

Como vimos anteriormente, se pueden leer caracteres individuales utilizando getchar() o una función derivada. También se puede utilizar scanf() para este propósito si se usa el especificador de formato %c.

Aunque cuando se lee otro tipo de datos, los espacios, las tabulaciones y los saltos de línea se usan como separadores de campo, cuando se leen caracteres individuales ese tipo de caracteres de espacio en blanco se leen como cualquier otro carácter. Por ejemplo, con una secuencia de entrada como "x y", este fragmento de código

```
scanf("%c%c%sc", &a, &b, &c);
```

coloca el carácter "x" en a, "el espacio" en "b" y el carácter "y" en c.

3.4.5. Lectura de cadenas

Se puede utilizar la función scanf() para leer una cadena de la secuencia de entrada utilizando el especificador de formato %s. Con %s se hace que scanf() lea caracteres hasta que encuentre un espacio en blanco. Los caracteres que se leen se van colocando en el arreglo (es simplemente una matriz) de caracteres apuntado por el argumento correspondiente, añadiéndose un terminador nulo al final. Por lo que se refiere a scanf(), un carácter de espacio en blanco es, un salto de línea, una tabulación, una tabulación vertical o un salto de página. Esto significa que no se puede utilizar scanf() para leer una cadena como "esto es una prueba", porque el primer espacio terminará el proceso de lectura.

Las alternativas que existen para esto son: gets() y puts(). La función gets() lee una cadena de caracteres introducida por el teclado y la coloca en la dirección apuntada por el argumento. Se pueden introducir caracteres por el teclado hasta que se pulse el retorno de carro. Hay que tener cuidado con gets() porque no lleva acabo una comprobación de limites sobre el arreglo que recibe la entrada . Así el usuario puede introducir más caracteres de los que caben en el arreglo y en ocasiones obtener resultados desastrosos. La función puts() escribe su argumento

de tipo cadena en la pantalla seguido de un carácter de salto de línea y reconoce los mismos códigos de barra invertida, tales como \n para salto de línea; puts() solo puede imprimir una cadena de caracteres; no puede imprimir números o realizar conversiones de formato. Por esto puts() es utilizado a menudo cuando no se requieren conversiones de formato.

Veamos un pequeño ejemplo:

Programa 4

```
/* Este programa nos muestra el uso de puts() y gets() */

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

void main(void)
{
  char cadena[20];

  clrscr();
  puts("Teclea una cadena (máximo 19 caracteres): ");
  gets(cadena); /*gets lee el espacio en blanco/
  puts("Lo que tecleaste fue: \a");
  puts(cadena);
  printf("Teclea una cadena (máximo 19 caracteres): ");
  scanf("%s", &cadena );/*scanf no lee el espacio en blanco*/
  printf("Lo que tecleaste fue: \a");
  printf("%s",cadena);
  getch();

}
```

El programa anterior ¿hace lo mismo con puts() y gets() que con printf() y scanf()? ó ¿cuál es la diferencia? La respuesta queda a iniciativa del lector. Basta con correr el programa una y otra vez. Si se quieren obtener mas variantes se puede modificar el código.

Obs. Es importante hacer notar que este programa es muy pequeño y tal vez no se alcancen a ver algunas consecuencias que pudiera arrojar si no se introducen adecuadamente las cadenas.

3.4.6. Lectura de una dirección

Para leer una dirección de memoria se utiliza el especificador de formato %p. Este especificador hace que scanf() lea una dirección en el formato utilizado por la arquitectura de la CPU. Veremos este mas a detalle en el capitulo 4.

3.4.7. El especificador %n

El especificador %n indica a scanf() que ha de asignar a la variable entera apuntada por el correspondiente argumento el número de caracteres leídos desde la secuencia de entrada (hasta el punto en el que se haya encontrado el especificador %n).

3.4.5. Modificadores de formato

Al igual que printf(), scanf() permite que se modifiquen algunos de sus especificadores de formato. Los especificadores de formato pueden incluir un modificador de longitud máxima de campo. Se trata de un entero, situado entre el % y el especificador de formato, que limita el número de caracteres leídos para ese campo. Por ejemplo, para no leer más de 20 caracteres en cad se escribirá:

```
scanf("%20s", cad);
```

Si la secuencia de entrada está formada por más de 20 caracteres, una llamada posterior comenzará donde la anterior se quedó. Por ejemplo, si se introduce

ABCDEFGHIJKLMNOPQRSTUVWXYZ

como respuesta a la llamada a scanf(), sólo se asignan a cad los 20 primeros caracteres, o lo que es lo mismo, hasta el carácter "T", debido al especificador de longitud máxima de campo. Esto significa que el resto de los caracteres, UVWXYZ, no se han utilizado todavía.

Si se realiza otra llamada a scanf() como

```
scanf("%s", cad);
```

se colocan las letras UVWXYZ en cad. La entrada de un campo puede terminar antes de que se alcance la longitud máxima si se encuentra un carácter de espacio en blanco. En ese caso, scanf() pasa al siguiente campo.

Para leer un entero largo se pone una l delante del especificador de formato. Para leer un entero corto se pone una h delante del especificador de formato. Estos modificadores se pueden usar con los códigos de formato d, i, o, u, x y n.

Por default, los especificadores f, e y g indican a scanf() que asigne el dato a una variable de tipo float. Si se pone una l delante de alguno de esos especificadores, scanf() asigna los datos a un double. Si se usa L se indica a scanf() que la variable que recibe el dato es de tipo long double.

3.5. Variables

Una variable es una localidad (posición) de memoria con nombre que se usa para mantener un valor que puede ser modificado por el programa. Todas las variables deben estar declaradas antes de poder ser utilizadas. La forma general de declaración es:

tipo lista _ de variables;

Aquí, tipo debe ser un tipo de datos válido con algún modificador y la lista_de variables puede consistir en uno o más nombres de identificadores separados por comas. A continuación se muestran algunas declaraciones:

```
int    i, j, l;
short int   si;
unsigned int   uí;
double    balance, beneficio, pérdida;
```

Obs. Recuerde que en C el nombre de una variable no tiene nada que ver con su tipo.

Existen tres tipos de variables que son:

- ✓ Variables locales
- ✓ Variables globales
- ✓ Parámetros formales

3.5.1. Variables locales

Las variables que se declaran dentro de una función se denominan variables locales. Las variables locales pueden ser utilizadas sólo en las instrucciones que estén dentro del bloque en el que han sido declaradas. O dicho de otra forma, las variables locales no son conocidas fuera de su propio bloque de código. Recuerda que un bloque de código comienza con una llave de apertura y termina con una llave de cierre.

Las variables locales sólo existen mientras se está ejecutando el bloque de código en el que son declaradas. O sea, la variable local se crea al entrar en el bloque y se destruye al salir de él. Más aún, una variable declarada dentro de un bloque de código no tiene que ver con otra variable con el mismo nombre que se haya declarado en otro bloque de código distinto.

El bloque de código más normal en el que se declaran variables locales es la función. Por ejemplo, considere las dos siguientes funciones:

```
void funcl (void)
{
  int x;      /* Esta variable es local porque esta declarada dentro de una función */
  z = 10 ;    /* La función es todo lo que esta entre la llaves */
}
void func2 (void)
{
 int x;
 z = -199;
}
```

- La variable entera z se declara dos veces, una vez en funcl() y otra en func2(). La z de funcl() no se corresponde ni tiene relación con la z de func2(). Tal como se dijo, cada z sólo es conocida en el propio código en que se ha declarado la variable.

Por conveniencia y tradición, la mayoría de los programadores declaran todas las variables que se necesitan en una función al principio del bloque de código de la misma, tras la llave de apertura y antes que cualquier instrucción (ver ejemplos anteriores). Sin embargo, se pueden declarar variables locales en cualquier bloque de código.

Cuando una variable definida en un bloque interno tiene el mismo nombre que una variable declarada en un bloque externo, la variable del bloque interno oculta la variable del bloque externo. Considérese lo siguiente:

Programa 5

```
/* Este programa muestra el uso de una variable declarada en un bloque especifico, siendo el caso
de que existe otra declarada al inicio del programa */

#include <stdio.h>

int main(void)
{
 int x;    /* x externa */
 x = 10 ;
 if (x == 10)
 {
   int x;       /* esta x oculta a la x externa */
   x = 99;
   printf("x interna: %d\n", x);
     }
       printf()("x externa: %d\n", x);
 return() ;
 }
```

Obs. Para este ejemplo consideremos el bloque del if (este bloque es lo que se encuentra entre las llaves que le preceden) como un conjunto de instrucciones; en el capítulo 3 sabremos cómo y para qué se usa.

El programa muestra lo siguiente:

- x interna 99
  x externa 10

- En este ejemplo, la x que se ha declarado en el bloque if interno oculta a la x externa. Por tanto, la x interna y la x externa son dos objetos independientes y distintos. Una vez que ese bloque termina, la x externa vuelve a ser visible.

En C89 se deben declarar todas las variables locales al principio del bloque en el que se usan, antes que cualquier instrucción "ejecutable". Por ejemplo, la siguiente función es técnicamente incorrecta y no compilará en un compilador compatible con C89.

```
/* Esta función es errónea si se compila como un programa C89.*/

void f(void)
{
  int i;
  i = 10;
  int j;        /* esta linea producirá un error */
  j= 20;
  }
```

Sin embargo, en C99 (y en C++), esta función es perfectamente válida ya que se pueden declarar variables locales en cualquier lugar de un bloque, previamente a su primer uso.

Dado que las variables locales se crean y se destruyen cada vez que se entra o se sale del bloque en el que son declaradas, su contenido se pierde tras salir del bloque. Es especialmente importante recordar esto por lo que respecta a las llamadas de funciones (en el capítulo cuatro profundizaremos sobre el uso de funciones).

Cuando se llama una función se crean sus variables locales, y cuando se sale de ella se destruyen. Esto significa que las variables locales no pueden retener sus valores entre llamadas. (Sin embargo, se puede indicar al compilador que retenga sus valores mediante el uso del modificador static).

3.5.2. Variables globales

A diferencia de las variables locales, las variables globales se conocen a lo largo de todo el programa y se pueden usar en cualquier parte del código. Además, mantienen sus valores durante toda la ejecución del programa. Las variables globales se crean al declararlas en cualquier parte fuera de una función. Pueden ser accedidas por cualquier expresión, independientemente de la función en la que se encuentre la expresión.

Si una variable global y una variable local tienen el mismo nombre, todas las referencias a ese nombre de variable dentro de la función donde se ha declarado la variable local se refieren a esa variable local y no tienen efecto sobre la variable global.

Las variables globales son muy útiles cuando se usan los mismos datos en varias funciones del programa. Sin embargo, se debe evitar el uso de variables globales innecesarias, el uso de un gran número de variables globales puede producir errores en el programa debido a efectos secundarios desconocidos y no deseables.

3.6. Cualificadores de acceso

C define cualificadores de tipo que controlan las formas en que se acceden o se modifican las variables. C89 define dos de esos cualificadores: const y volatile. (C99 añade un tercero, denominado restrict). Los cualificadores de tipo deben preceder a los nombres de tipo que cualifican.

3.6.1 . const

Las variables de tipo const no pueden ser modificadas por el programa Una variable const recibe su valor bien por una inicialización explícita o bien por algún medio dependiente del hardware, una variable de tipo const puede ser modificada por algo externo al programa.

Por ejemplo, por un dispositivo hardware que establezca su valor. Sin embargo, declarándola como const se puede probar que cualquier cambio en la variable se debe a sucesos externos.

3.6.1.1. Constantes hexadecimales y octales

A veces es más cómodo usar un número en base 8 ò 16, que en base 10. El sistema de numeración en base 8 se denomina octal y usa los dígitos del 0 al 7. El número 10 en octal equivale al 8 en decimal. El sistema de numeración en base 16 se denomina hexadecimal y usa los dígitos del 0 al 9 y las letras de la A a la F, que representan, respectivamente, los valores 10, 11, 12, 13, 14 y 15. Por ejemplo, el número 10 en hexadecimal equivale al 16 en decimal. Dado que estos sistemas de numeración se utilizan frecuentemente, C permite especificar constantes enteras en hexadecimal o en octal en lugar de en decimal. Una constante hexadecimal consiste en Ox seguido de la constante en forma hexadecimal. Una constante octal comienza por O. Aquí van unos ejemplos:

```
int hex = 0x80;        /* 128 en decimal */
int oct = 012;         /* 10 en decimal */
```

3.6.1.2. Constantes de cadena

C contempla otro tipo de constante: la cadena. Una cadena es una secuencia de caracteres encerrados en comillas dobles. Por ejemplo, "esto es una prueba" es una cadena. Ya han aparecido ejemplos de cadenas en algunas de las instrucciones printf() de los programas de ejemplo. Aunque C permite definir constantes de cadena, no tiene formalmente un tipo de datos cadena.

No deben confundirse las cadenas con los caracteres. Una constante de carácter irá entre comillas simples, tal como `a'. Sin embargo, "a" es una cadena que contiene sólo una letra.

3.4.1.3. Secuencias de Escape

El incluir entre comillas simples las constantes de carácter es suficiente para la mayoría de los caracteres imprimibles. Pero unos pocos, como el retorno de carro, son imposibles de introducir desde el teclado. Por esta razón, C incluye el caso especial de constantes *de car*ácter con barra invertida, que se muestran en la **Tabla 3.7.** Para facilitar la colocación de esos caracteres especiales como constantes. También se conocen como secuencias de escape. Se deben usar los códigos de barra invertida en lugar de sus equivalentes ASCII con el fin de asegurar la portabilidad del código.

Tabla 3.7 Secuencias de Escape.

Código	Significado
✓ \b	Espacio atrás.
✓ \f	Salto de página.
✓ \n	Salto de línea.
✓ \r	Retorno de carro.
✓ \t	Tabulación horizontal.
✓ \"	Comillas dobles.
✓ \'	Comilla simple.
✓ \\	Barra invertida.
✓ \v	Tabulador vertical.
✓ \a	Despliega un sonido de Alerta.
✓ \?	Signo de interrogación.
✓ \N	Constante octal (donde N es una constante octal).
✓ \xN	Constante hexadecimal (donde N es una constante hexadecimal).

A continuación se presenta un ejemplo que ilustra la utilidad de las secuencias de escape.

Programa 6

```
/* Este programa exhibe como imprimir algunos caracteres que no se pueden introducir mediante el
teclado */

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

void main()
{
  clrscr();

  printf("Campana\aCampana\aCampana\a\n");
  printf("ABCDEFGH");
  printf("\b\b\b\b\bXY\n");
  printf("Pagina1\fPagina2\fPagina3\n");
  printf("A\nB\nC\n");
  printf("abcdefgh \rXY\n");
  printf("A\tB\tC\n");
  printf("C:\\TEMP\CURSO\n");
  printf("\"Lenguaje C\"\n");
  printf("La \'x\' es un carácter\n");
  printf("\n\tCampana\a\nCampana\a\tCampana\a\a\a\n\n");
}
```

La salida de este programa es:

```
(Inactive C:\TCWIN\BIN\PRO...
CampanaCampanaCampana
ABCXY
Pagina1■Pagina2■Pagina3
A
B
C
abcdefgh
XY
A       B       C
C:\TEMPCURSO
"Lenguaje C"
La 'x' es un caracter

        Campana
Campana Campana
```

- Aquí lo importante es observar como se coloca cada una de las secuencias de escape dentro de printf() y la salida que genera. Es muy fácil si vas siguiendo línea a línea el programa y vas verificando qué hace, puedes ayudarte con la tablita anterior que describe cada una de éstas.

3.6.2. Volatile

El cualificador volatile indica al compilador que el valor de una variable puede cambiar por medios no explícitamente especificados por el programa. Por ejemplo, la dirección de una variable global puede ser pasada a la rutina de reloj del sistema operativo y usada para mantener el tiempo real del sistema. En esta situación los contenidos de esa variable son alterados sin una instrucción de asignación explícita del programa. Esto es importante porque la mayoría de los compiladores de C automáticamente optimizan ciertas expresiones asumiendo que el contenido de una variable no cambia si no aparece en la parte izquierda de una instrucción de asignación; así puede que no se vuelva a comprobar la variable cada vez que se referencia. Además, algunos compiladores cambian el orden de evaluación de una expresión durante el proceso de compilación. El cualificador volatile previene que se hagan esos cambios.

3.7. Operadores

C es un lenguaje muy rico en operadores incorporados. De hecho, dota de un mayor significado a los operadores que la mayoría de los demás lenguajes de programación. Hay cuatro clases principales de operadores: aritméticos, relacionales, lógicos y en el ámbito de bits. Hay además algunos operadores especiales, como el operador de asignación, para otras tareas particulares.

3.7.1. El operador de asignación

La forma general del operador de asignación es:

```
nombre de variable = expresión;
```

donde la expresión puede ser tan simple como una constante o tan compleja como se requiera C.

Por ejemplo:

```
Suma = 10;
Contador = 12+65/cos(2)+tag(X)*3.1416;
```

3.7.1.1. Asignaciones múltiples

Se puede asignar a muchas variables el mismo valor utilizando asignaciones múltiples en una sola instrucción. Por ejemplo:

```
a=b=c=0;
```

que es lo mismo que poner

```
a=0;
b=0;
c=0;
```

En los programas profesionales a menudo se usa el de asignaciones múltiples para asignar valores comunes a las variables.

3.7.1.2. Asignaciones compuestas

Existe una variante del uso del operador de asignación que se denomina *asignación* compuesta y que simplifica la escritura de ciertos tipos de asignaciones. Por ejemplo:

```
x=x+10;
```

puede escribirse como

```
x+=10;
```

El operador + = indica al compilador que ha de asignar a x el valor de x más 10. Existen operadores de asignación compuesta para todos los operadores binarios (los que requieren dos operandos). En general, una expresión como:

variable = variable operador expresión;

se puede escribir como

variable operador = expresión;

Como la asignación compuesta es más corta que el = equivalente, a menudo se refieren como abreviaturas de asignación. La asignación compuesta es muy utilizada en los programas en C profesionales, por lo que conviene familiarizarse con ella.

Veamos un ejemplo que ilustra lo anterior.

Programa 7

```
/* Este programa nos ejemplifica como utilizar una asignación compuesta */

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

void main()
{
  int valor =100;

  clrscr();
  printf("Valor INICIAL=%d\n",valor);

  valor/=10;  /* valor = valor/10 */
  printf("después de DIVISION, valor=%d\n",valor);

  valor-=5;   /* valor = valor-10 */
  printf("Después de la RESTA, valor=%d\n",valor);

  valor*=5;   /* valor = valor*10 */
  printf("Después de la MULTIPLICACION, valor=%d\n", valor);

  valor+=15;  /* valor = valor+15 */
  printf("Después de la SUMA, valor=%d\n",valor);

  valor%=10;  /* valor = valor%10 */
  printf("Después del MODULO, valor=%d", valor);

  getch();

}
```

La salida de este programa es:

```
C:\TCWIN\BIN\NONAME00.EXE
Valor INICIAL=100
después de DIVISION, valor=10
Después de la RESTA, valor=5
Después de la MULTIPLICACION, valor=25
Después de la SUMA, valor=40
Después del MODULO, valor=0
```

- Aquí podemos observar cómo funcionan las abreviaturas de asignación y como va cambiando el contenido de la variable valor.

Los operadores +, -, * y / operan tal y como los conocemos en la Matemáticas. El operador % de módulo obtiene el resto (residuo) de una división entera.

3.7.2. Los operadores de incremento y decremento

C contiene dos operadores muy útiles que simplifican dos operaciones habituales. Son el de incremento y el de decremento, ++ y -- respectivamente. El operador ++ añade 1 a su operando y -- le resta 1.

Veamos un ejemplo de estos últimos.

Programa 8

```
/* Este programa nos muestra el uso correcto de los operadores incremento y decremento */

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

void main()
{
  int m=5,n=5;
  int x=0,y=0;
  clrscr();
  printf("Valores iniciales :m= %d ,n= %d ,x= %d ,y= %d\n\n", m, n, x, y);
  x=++m;  /* Preincremento */
  y=n++;  /* Postdecremento */
  printf("Después de {x=++m} y {y=n++}\n");
  printf("Valores actuales : m=%d ,n=%d , x=%d , y=%d\n\n", m, n, x, y);
  x= --m;  /* Predecremento */
  y=n--;  /* Posdecremento */
  printf("Después de {x=--m} y {y=n--}\n");
  printf("Valores finales : m=%d ,n=%d , x=%d , y=%d\n\n", m, n, x, y);
  getch();
}
```

La salida de este programa es:

```
C:\TCWIN\BIN\PROG8.EXE
Valores iniciales :m= 5 ,n= 5 ,x= 0 ,y= 0

Despues de {x=++m} y {y=n++}
Valores actuales : m=6 ,n=6 , x=6 , y=5

Despues de {x=--m} y {y=n--}
Valores finales : m=5 ,n=5 , x=5 , y=6
```

- Aquí podemos observar que no es lo mismo ++n que n++; en el primer caso incrementamos el valor antes de imprimirlo y en el segundo después de imprimirlo (i.e. se imprime el mismo valor de la variable pero el contenido de la variable cambia).Algo análogo sucede con –n y ++n.

Obs. En el siguiente capítulo cuando veamos ciclos retomaremos estos operadores y profundizaremos más en su uso y entendimiento.

3.7.3. Operadores relacionales y lógicos

En C, cierto es cualquier valor distinto de cero. Las expresiones que utilizan los operadores relacionales y lógicos devuelven 1 para cierto y 0 para falso. La **Tabla 3.8** muestra los operadores relacionales y lógicos.

Tabla 3.8

P	*Q*	*P && q*	*P || q*	*!p*
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Tanto los operadores relacionales como los lógicos tienen un nivel de precedencia menor que los operadores aritméticos. Esto significa que una expresión como 10>1+8 se evalúa como si se hubiera escrito 10 >(9).

3.7.3.1. Operadores relacionales

Tabla 3.9

Operador	Comparación
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
= =	Igual
!=	Distinto

3.7.3.2 Operadores lógicos

Tabla 3.10

<table>
<tr><th>Operador</th><th>Acción</th></tr>
<tr><td>&&</td><td>Conjunción</td></tr>
<tr><td>||</td><td>Disyunción</td></tr>
<tr><td>!</td><td>Negación</td></tr>
</table>

La tabla **3.11** muestra la precedencia relativa entre los operadores relacionales y lógicos:

Tabla 3.11

<table>
<tr><td>Mayor precedencia</td><td>!</td></tr>
<tr><td></td><td>> >= < <=</td></tr>
<tr><td></td><td>= = ¡=</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td>Menor precedencia</td><td>||</td></tr>
</table>

3.7.4. Operadores a nivel de bits

Al contrario que muchos otros lenguajes, C contiene un completo juego de operadores a nivel de bits. Dado que C se diseñó para sustituir al lenguaje

ensamblador en muchas tareas de programación, era importante permitir todas las operaciones que se pueden realizar en ensamblador. Este tipo de operadores no serán tratados en este trabajo pero es importante saber que existen y que en un momento dado se puede contar con ellos.

3.7.5. El operador ?

C contiene un operador muy potente y conveniente que puede usarse para sustituir ciertas instrucciones de la forma if-then-else. El operador ternario ? toma la forma general

```
Expl ? Exp2 : Exp3;
```

donde:

Expl, Exp2 y Exp3 son expresiones.

Obsérvese la posición de los dos puntos. El operador ? actúa de la siguiente forma: evalúa Expl. Si es cierta, evalúa Exp2 y toma ese valor como resultado de toda la expresión. Si Expl es falsa, evalúa Exp3 tomando su valor como resultado de toda la expresión. Por ejemplo, en

```
Numero1 = 10;
Numero2 = Numero1>9 ? 100 : 200;
```

A Número2 se le asigna el valor 100. Si Numero1 hubiera sido menor que 9, habría recibido el valor 200. El operador ? se tratará más detenidamente en los capítulos posteriores.

*3.7.6. Los operadores de apuntadores & y **

Un apuntador es la dirección de memoria de una variable. Una variable apuntador es una variable específicamente declarada para contener un apuntador a su tipo especifico. Los apuntadores son una de las carácterísticas más potentes de C, siendo utilizados para una gran variedad de propósitos. Por ejemplo, proporcionan una rápida forma de referenciar los elementos de un arreglo (matriz). Permiten a las funciones modificar los parámetros de llamada. Dan soporte a las listas enlazadas y a otras estructuras de datos dinámicas.

El primer operador de apuntadores es &, un operador unario que devuelve la dirección de memoria del operando. (Recuérdese que un operador unario es aquel que sólo requiere un operando). Por ejemplo:

```
mes = &cont;
```

coloca en mes la dirección de memoria de la variable cont. Ésta es la dirección de la posición interna en la computadora de la variable. No tiene nada que ver con el valor de cont. Se puede pensar en & como significando "la dirección de". Por tanto, la instrucción anterior de asignación significa "mes recibe la dirección de cont".

Para comprenderlo mejor, supongamos que la variable cont utiliza la posición de memoria 2000 para guardar su valor. También supongamos que el valor de cont es 100. Después de la asignación, mes tendrá el valor 2000.

El segundo operador de apuntadores es *, que es el complementario de &. Es un operador unario que devuelve el valor de la variable ubicada en la dirección que se especifica. Por ejemplo, si mes contiene la dirección de memoria de la variable cont, entonces

```
q = *mes;
```

colocará el valor de cont en q. Ahora q tendrá el valor 100 ya que 100 es lo guardado en la posición 2000, que es la dirección de memoria que contiene mes. Piénsese en * como significando "en la dirección". En este caso, la instrucción de asignación significa "q recibe el valor en la dirección mes".

Por desgracia, el signo de la operación Y a nivel de bits y el de "la dirección de" son el mismo y el de multiplicación y el de "en la dirección" también. Esos operadores no tienen relación entre sí. Tanto & como * tienen una precedencia mayor que cualquier operador aritmético, excepto el menos unario, respecto del cual la tienen igual.

Las variables que vayan a contener apuntadores se han de declarar como tales, colocando un * delante del nombre de la variable. Esto indica al compilador que va a contener un apuntador a ese tipo de variable. Por ejemplo, para declarar cont como apuntador a carácter podemos escribir:

```
char *cont;
```

Es importante comprender que aquí cont no es un carácter, sino un apuntador a un carácter; hay una gran diferencia. El tipo de dato al que apunta un apuntador, en este caso char, se denomina tipo *base* del apuntador. Sin embargo, la propia variable apuntador es una variable que mantiene la dirección de un objeto del tipo base. Así, un apuntador a carácter (o cualquier apuntador, generalizando) tiene un tamaño suficiente para guardar una dirección tal como esté definida por la arquitectura de la computadora que se utilice. Es el tipo base el que determina lo que puede haber en esa dirección.

Se pueden mezclar variables apuntador y normales en la misma instrucción de declaración.

Por ejemplo,

```
int x, *y, cont;
```

declara "x" y "cont" como de tipo entero e y como apuntador a un dato entero.

A continuación se utilizan los operadores & y * para poner el valor 10 en la variable destino. Como es de esperar, este programa muestra en la pantalla el valor 10.

Programa 9

```
/*Este programa realiza el paso de un valor de una variable a otra utilizando una variable puntero. */

#include <stdio.h>

int main(void)
{
  int destino, origen;
  int *m;
  origen = 10;
  m = &origen;
  destino = *m;
  printf("%d", destino);
  return 0;
  }
```

3.7.7. La coma como operador

Como operador, la coma encadena varias expresiones. La parte izquierda del operador coma siempre se evalúa como void. Esto significa que la expresión de la parte derecha se convierte en el valor de la expresión total separada por coma. Por ejemplo,

```
x = (y=3, y+1);
```

primero asigna el valor 3 a "y" y luego asigna el valor 4 a x. Los paréntesis son necesarios debido a que el operador coma tiene menor precedencia que el operador de asignación.

Esencialmente, la coma provoca una secuencia de operaciones. Cuando se usa en la parte derecha de una instrucción de asignación, el valor asignado es el valor de la última expresión de la lista separada por comas.

De alguna forma se puede pensar en el operador coma como teniendo el mismo significado que la palabra "y" en español, como en la frase "haz esto y esto y esto".

3.7.8. Los operadores [ ] y ( )

Los paréntesis son operadores que aumentan la precedencia de las operaciones que contienen. Los corchetes llevan a cabo el indexamiento de arreglos. Dado un arreglo, la expresión entre corchetes proporciona un índice para el arreglo. Por ejemplo:

Si tenemos el siguiente arreglo

```
char c[80];
```

la siguiente instrucción

```
c[3]= 'x';
```

asigna primero el valor `x' al cuarto elemento (téngase en cuenta que todos los arreglos empiezan en la posición cero) del arreglo c.

3.8. Expresiones

Operadores, constantes y variables son lo que constituyen las expresiones. Una expresión en C es cualquier combinación válida de esos elementos. Como la mayoría de las expresiones tienden a seguir las reglas generales del álgebra, a menudo se dan por sabidas. Sin embargo, hay algunos aspectos que son específicos del lenguaje C.

3.9. Moldes

Se puede forzar que una expresión sea de un tipo determinado utilizando una construcción denominada molde(en inglés cast".) La forma general de un molde es

```
(tipo) expresión
```

donde tipo es un tipo de datos válido. Por ejemplo, si se quiere asegurar que la expresión x/2 se evalúe como de tipo float, se puede escribir

```
(float) x/2
```

Técnicamente, los moldes son operadores. Como operador es unario y tiene la misma precedencia que cualquier otro operador unario.

Capítulo 4
Condicionales y Ciclos

En su sentido más general, una expresión es una parte del programa que se puede ejecutar. Es decir, una instrucción especifica una acción. C clasifica las instrucciones en estos grupos:

- ✓ Selección.
- ✓ Iteración.
- ✓ Salto.
- ✓ Etiquetado.
- ✓ Expresión.
- ✓ Bloque.

Entre las instrucciones de selección se encuentran if() y switch(). (A menudo se usa el término instrucción condicional en lugar de instrucción de selección). Las instrucciones de iteración son while(), for() y do-while(). También se conocen como instrucciones de ciclos. Las instrucciones de salto son break, continue, go to y return. Las instrucciones de etiquetado son case y default (que se tratan junto con la instrucción switch()) y la propia label (que se trata junto con go to). Las instrucciones de expresión son las compuestas por expresiones válidas. Las instrucciones de bloque son simplemente bloques de código. (Un bloque comienza con "{" y acaba en "}".) Las instrucciones de bloque también se conocen como instrucciones compuestas.

Como muchas de las instrucciones basan su funcionamiento en el resultado de alguna prueba condicional, empezaremos repasando los conceptos de verdadero y falso.

4.1. Instrucciones de bloque

Las instrucciones de bloque son simplemente grupos de instrucciones relacionadas que se tratan como una unidad. Las instrucciones que componen un bloque quedan agrupadas de forma lógica. Las instrucciones de bloque también se denominan instrucciones compuestas. Un bloque comienza con una llave { ,y termina con su pareja }. Los programadores usan los bloques de instrucciones normalmente para crear objetivos multiinstrucción de algunas otras instrucciones, tales como el if(). Sin embargo, se puede colocar un bloque de instrucciones en cualquier sitio en que se pueda poner una instrucción. Por ejemplo lo que sigue es perfectamente valido en C aunque resulte inusual.

```
#include <stdio.h>

int main(void)
{

int i ;
{        /* Una instrucción de  bloque */

        i=120;
printf( " %d" , i);
}

return 0;

}
```

4.2. Verdadero y Falso en C

Muchas instrucciones de C se basan de una expresión lógica que determina la acción que se ha de llevar a cabo. Una expresión condicional tiene como resultado un valor cierto o falso. En C, cualquier valor distinto de cero es cierto, incluyendo los números negativos. El 0 es el único valor falso. En este enfoque de lo que es cierto y lo que es falso permite codificar de forma extremadamente eficiente muchos tipos de rutinas.

4.3. Instrucciones de Selección

C contempla dos tipos de instrucciones de selección: if() y switch(). Además, el operador ? es una alternativa para if() en ciertas situaciones.

4.3.1. if()

La forma general de la instrucción if() es:

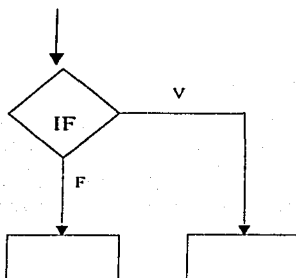
```
if (expresión)
        instrucción;
else
        instrucción;
```

donde instrucción puede ser una instrucción simple, un bloque de instrucciones o nada (en el caso de instrucciones vacías). La cláusula else es opcional.

Si la expresión es cierta (cualquier valor que no sea 0), se ejecuta la instrucción o el bloque de instrucciones que constituyen el cuerpo del if(); en caso contrario se ejecuta la instrucción ó el bloque de instrucciones que constituye el cuerpo del else, si existe. Recuérdese que sólo se ejecuta el código asociado al if() o al else, nunca ambos.

Podemos ver lo anterior en **el Diagrama 4.1.**

Diagrama 4.1



- **Un diagrama como el 4.1. es conocido como diagrama de flujo y es una representación grafica de un algoritmo. Las flechas nos indican el camino que debemos seguir, el rombo nos permite tomar una decisión mediante un condicionante que solo acepta como respuestas Falso o Verdadero y en cada uno de los rectángulos se realiza una tarea especifica.**
- **En este diagrama podemos observar dos opciones la Falsa (F) y la verdadera (V). La verdadera equivale a que se cumple la condición del if() y se realiza lo correspondiente a este bloque; la falsa corresponde al else en el caso de que este exista ó a salir del if() si no existe este.**

La expresión condicional que controla el if() debe producir un resultado escalar. Un escalar es cualquiera de los tipos entero, real, apuntador o de carácter. (En C99, Bool es también un tipo escalar, pudiéndose utilizar también en la expresión del if()) No es habitual usar un numero real para controlar una instrucción condicional ya que disminuye la velocidad de ejecución considerablemente. Se necesitan varias instrucciones para realizar una operación en punto flotante. Se necesitan relativamente pocas instrucciones para llevar a cabo una operación entera o de carácter.

Veamos algunos ejemplos con if().

Programa 10

```
/* En este programa si el numero introducido mediante el tecleado se encuentra entre 1 y 10 lo eleva
a la cuarta potencia  de lo contrario termina el programa */

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

void main(void)
{

 int numero;
  clrscr();
  printf("Teclea un numero entre 1 y 10: ");
  scanf("%d", &numero);
  if(numero >= 1  && numero <= 10)
  {
    printf("\nEl numero %d elevado a la cuarta potencia es %lf", numero, pow(numero, 4));
    printf("\n\nY la raiz cuadrada en exponencial es %lE", sqrt(numero));
  }
  getch();

}  numero
```

- Este programa nos pide que tecleemos un numero entre el 1 y el 10; este numero lo guardamos en la variable numero.
- Cuando realizamos la validación con el if() si el numero esta entre el 1 y el 10 se ejecuta el bloque de instrucciones, de lo contrario no y automáticamente salta a la siguiente instrucción, que en este caso es getch(). Puedes probar lo anterior corriendo el programa varias veces y dándole diferentes números.

Programa 11

```
/* Este programa nos pide que ingresemos dos números y divide  el primero entre el segundo */

# include <stdio.h>

void main(void)
{
  int numero1, numero2;

  printf("\nIntroduzca dos números [n1,n2]: ");
  scanf("%d,%d", &numero1, &numero2);
```

```
  if(numero2)
    printf("\nLa división de los números es: %d\n", numero1/numero2);
  else
    printf("\nNo se puede dividir por cero \n");
}
```

- En el programa anterior se nos pide que introduzcamos 2 números, los cuales deben de estar separados por comas. ¿qué pasa si los separo con espacios o con cualquier otro tipo de carácter? ¿Por qué deben ir separados por comas? ¿ Donde indica que deben ir separados con comas?

- Una vez que introduzco los 2 números los guardo en la variable1 y variable2 respectivamente. Posteriormente el if() verifica que numero 2 sea diferente de cero; si es diferente de cero ejecuta el bloque del if(), de lo contrario ejecuta el bloque del else.

- En el programa anterior las variables han sido declaradas como enteros, por lo que se espera que los números introducidos sean enteros; ¿qué pasa si introducimos un real, un carácter o un numero de otro tipo?

- Como habrá notado este programa divide al numero1/numero2 y como la división de dos enteros es entera (para el lenguaje C), el resultado va a ser un entero (al resultado de la división le va a truncar la parte fraccionaria, no va a redondear el numero como se esperase que lo hiciera).

Obs. Recordemos que para C todo numero diferente de cero es verdadero y el cero es falso.

Existen maneras de validar la entrada de datos y así poder evitar posibles desastres en nuestros programas.

4.3.1.1. if() anidados

Un if() anidado es un if() que es el cuerpo de otro if() o else. Los if() anidados son muy comunes en programación. En un if() anidado, una instrucción else siempre se refiere al if() más próximo que esté en el mismo bloque que el else y que no esté ya asociado con un else.

C89 especifica que el compilador debe permitir al menos 15 niveles de anidamiento. C99 eleva ese límite a 127. En la práctica, la mayoría de los compiladores permiten bastantes niveles más. Sin embargo, rara vez es necesario pasar de unos pocos niveles de anidamiento y un anidamiento excesivo puede rápidamente ensombrecer el significado de un algoritmo.

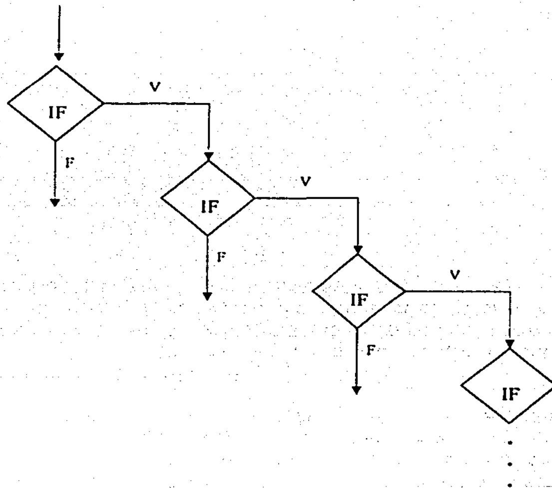
4.3.1.2. La escala if-else-if

La escala if-else-if se escribe generalmente como sigue:

```
if(expresión)
        instrucción;
        else
                if(expresión)
                        instrucción;
                        .
                        .
                        .
                        else
                                instrucción;
```

Observemos lo anterior en el **Diagrama 4.2.**

Diagrama 4.2



➤ **En este diagrama podemos observar que por cada elección de verdadero, tenemos otras dos opciones y así sucesivamente hasta encontrar una elección falsa. Al momento de encontrar esta última salimos automáticamente de todos los if() a los que habíamos ingresado.**

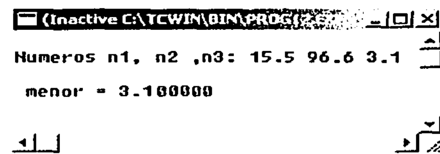
Programa 12

```
/* Este programa nos pide que se ingresen tres números y nos muestra el menor de ellos */

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

void main(void)
{

  float n1, n2, n3, menor;

  printf("\nNúmeros n1, n2 ,n3: ");
  scanf("%f %f %f", &n1, &n2, &n3);

  if (n1 < n2)
  {
    if (n1 < n3)
           menor=n1;
    else
           menor=n3;
  }
  else
  {
    if (n2 < n3)
           menor=n2;
    else
           menor=n3;
  }
  printf("\n menor = %g \n", menor);

}
```

- Este programa nos da el menor de tres números; para lo cual nos pide que los introduzcamos, posteriormente los almacena en las variables n1, n2, n3. En esta ocasión los números son flotantes y deben ir separados por blancos ¿por que?. . . la clave esta en los modificadores de formato del scanf() (compáralos con el programa11).

- Vamos a ver como trabajan los if() anidados: El primer if() verifica si n1<n2; si es menor ejecuta el if() de lo contrario ejecuta el else. Si ejecuto el if(), verifica con el siguiente if() si ni<n3 y se cumple la condición concluimos que n1 es el menor de los tres de lo contrario n3 será el menor. Si se ejecuto el else, se verifica si n2<n3, si se cumple la condición concluimos que n2 es el menor de lo contrario el menor será n3. Algo similar a esto se hace en Teoría de Juegos cuando se construyen los árboles de decisión.

- Observa con atención y ve siguiendo paso a paso la secuencia de los if() es muy fácil y con la práctica los dominarás.
- Una salida del programa introduciendo los números 15.5,96.6,3.1 es:



4.4. La alternativa ?

El uso del operador "?" para sustituir las instrucciones if-else no está limitado únicamente a instrucciones de asignación. Recuérdese que todas las funciones (excepto aquellas declaradas de tipo void) devuelven un valor. Por tanto, se pueden utilizar una o más llamadas a funciones en una expresión "?" Cuando se encuentra el nombre dé una función, se ejecuta esa función de modo que pueda determinarse su valor de vuelta. Por tanto, es posible ejecutar una o más llamadas a funciones al utilizar el operador "?", situándolas en las expresiones que constituyen sus operandos. Esto lo veremos más a fondo cuando veamos funciones.

4.5. La expresión condicional

A veces los que se encuentran por primera vez con el lenguaje C se extrañan de que se pueda usar cualquier expresión válida para controlar el if() y el operador "?" Es decir, no se está restringido a expresiones que utilicen los operadores lógicos y relacionales (como en el caso de lenguajes como BASIC o Pascal). El programa simplemente ha de evaluar la expresión a un valor cero o distinto de cero; ejemplo:

```
If(b!=0)
   printf("%d\n", a/b);
```

4.6. Switch()

C incorpora una instrucción de selección múltiple, denominada switch(), que compara sucesivamente el valor de una expresión con una lista de constantes enteras o de carácteres. Cuando se encuentra una correspondencia, se ejecutan las instrucciones asociadas con la constante. La forma general de la instrucción switch() es:

```
switch (expresión)
{
case constante1:
        secuencia de instrucciones
        break;
case constante2:
        secuencia de instrucciones
        break;
case constante3:
        secuencia de instrucciones
        break;
        .
        .
        .
default:
    secuencia de instrucciones
}
```

La expresión debe dar como resultado un valor entero. Por tanto, se pueden utilizar valores enteros o de carácter, mientras que las expresiones reales en coma flotante, por ejemplo, no están permitidas. Se comprueba el valor de la expresión, por orden, con los valores de las constantes especificadas en las instrucciones case. Cuando se encuentra una correspondencia, se ejecuta la secuencia de instrucciones asociada con ese case, hasta que se encuentra la instrucción break o el final de la instrucción switch(). La instrucción default se ejecuta si no se ha encontrado ninguna correspondencia. La instrucción default es opcional y si no está presente no se ejecuta ninguna acción al fallar todas las comprobaciones.

C89 especifica que una instrucción switch() debe poder tener como poco 257 instrucciones case. En la práctica, se suele limitar el numero de instrucciones case a muchas menos por razones de eficiencia. Aunque case es una instrucción de etiqueta, no tiene entidad por sí misma fuera de un switch().

La instrucción break es una de las instrucciones de salto de C. Se puede usar en los ciclos y en el switch(). Cuando se encuentra en un switch(), la ejecución del programa "salta" a la línea de código siguiente a este bloque de instrucciones. Podemos ver lo anterior en el **Diagrama 4.3.**

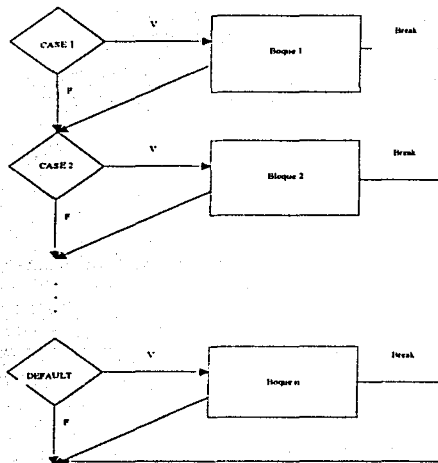
Hay tres cosas importantes que se deben saber sobre la instrucción switch():

- ✓ La instrucción switch() se diferencia de la instrucción if() en que switch() sólo puede comprobar la igualdad, mientras que if() puede evaluar expresiones relacionales o lógicas.

- ✓ No puede haber dos constantes case en el mismo switch() que tengan los mismos valores. Por supuesto, una instrucción switch() contenida en otra instrucción switch() puede tener constantes case que sean iguales.

- ✓ Si se utilizan constantes de tipo carácter en la instrucción switch(), se convierten automáticamente a sus valores enteros (tal como especifican las reglas de conversión de tipos de C).

Diagrama 4.3



- ➢ **Podemos observar como recorre cada uno de los case independientemente de si se cumple o no la condición (i.e. puede ejecutar todos los case, si no existe ningún break en alguno de ellos).**

- ➢ **Si se incluye la opción por default al menos ejecutara ésta; de lo contrario no ejecutara ninguna.**

- ➢ **Si se encuentra con algún break en un case automáticamente sale del switch().**

La instrucción switch() a menúdo se utiliza para procesar órdenes introducidas por teclado, como la orden de selección de un menú.

Técnicamente, las instrucciones break dentro de la instrucción switch() son opcionales. Se utilizan para finalizar la secuencia de instrucciones asociada con cada constante. Si se omite la instrucción break, la ejecución continúa en la siguiente instrucción case hasta que se alcanza una instrucción break o el final de switch().

Las dos facetas de la instrucción switch(). son:

- ✓ Que se pueden tener instrucciones case que no tengan secuencias de instrucciones asociadas. Cuando esto ocurre, la ejecución simplemente pasa al siguiente case.

- ✓ Y que la ejecución de una secuencia de instrucciones continúa en el siguiente case si no hay ninguna instrucción break.

El hecho de que los case se puedan ejecutar juntos cuando no hay break previene la innecesaria duplicidad de código, con lo que se obtiene un código muy eficiente.

Veamos un bonito ejemplo:

Programa 13

```
/* Programa de cambio de base - sentencia switch()
Cambia de la base --> a la base
          decimal --> hexadecimal
      hexadecimal --> decimal
          decimal --> octal
            octal --> decimal
*/
# include <stdio.h>
# include <conio.h>
# include <ctype.h>
void main(void)
{
char opción;
  int valor;
  printf("Transformar:\n");
  printf("\t\t 1: decimal en hexadecimal\n");
  printf("\t\t 2: hexadecimal en decimal\n");
  printf("\t\t 3: décimal en octal\n");
  printf("\t\t 4: octal en décimal\n");
```

```
printf("Escriba su opción: ");
opción=toupper(getche()); /* touuper convierte a mayúsculas el carácter */
putchar('\n');

switch(opción)
{
 case '1': printf("Introduzca un valor en decimal: ");
          scanf("%d", &valor);
          printf("%d en hexadecimal es %#x", valor, valor);
          break;

 case '2': printf("Introduzca un valor en hexadecimal: ");
          scanf("%x", &valor);
          printf("%x en decimal es %d", valor, valor);
          break;

 case '3': printf("Introduzca un valor en decimal: ");
          scanf("%d", &valor);
          printf("%d en octal es %#o", valor, valor);
          break;

 case '4': printf("Introduzca un valor en octal: ");
          scanf("%o", &valor);
          printf("%o en decimal es %d", valor, valor);
          break;
 }

}
```

- Este programa hace la conversión de números entre las bases decimal, octal y hexadecimal.

- Primero que nada despliega un menú con las opciones de conversión que tenemos. Y una vez elegida una opción nos pide el numero en determinada base y nos da como resultado su conversión.

- Cuando escogemos una opción el programa busca en cada case esta opción y si la encuentra ejecuta el case; y como todos los case como ultima instrucción tienen un break y no hay otra instrucción después del switch() el programa termina. ¿Qué pasa si tecleamos la opción 5? ¿Cuál es la alternativa para que el case encuentre una opción siempre?... es fácil ... piénsalo ... más adelante lo veremos.

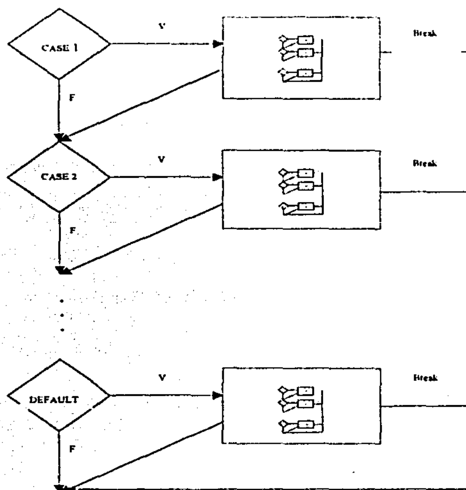
- Una posible salida del programa es la siguiente:

```
(Inactive C:\TCWIN\BIN\PROG13.EXE)
Transformar:
                      1: decimal en hexadecimal
                      2: hexadecimal en decimal
                      3: decimal en octal
                      4: octal en decimal
Escriba su opcion: 1
Introduzca un valor en decimal: 100
100 en hexadecimal es 0x64
```

4.6.1. Instrucciones switch() anidadas

Se puede tener un switch() formando parte de la secuencia de instrucciones de otro switch(). Incluso aunque haya constantes case del switch() interior y del exterior con valores iguales, no aparecen conflictos. Lo anterior se ilustra mejor en el **Diagrama 4.4** y se complementa con el programa 14.

Diagrama 4.4



- **En este diagrama podemos observar como dentro de cada case del switch() se puede realizar otro switch() totalmente independiente. A la salida de cada uno de estos switch() el switch() anterior pasa al siguiente case.**

Programa 14

```
/* Este programa crea una sencilla base de datos de vendedores que nos es útil para realizar
consultas de estos */

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

void main(void)

{
  char zona, vendedor;

  printf("\nLas zonas son: Este, Centro y Oeste\n");
  printf("Escriba la primera letra de la zona: \n");
  printf("Escriba su opción: ");
  zona=getche();
  zona=toupper(zona); /* se pasa a mayúsculas */
  putchar('\n');

  switch(zona)
  {
   case 'E':  printf("Los vendedores son Romualdo, Jeremias y Maria\n");
              printf("Escriba la primera letra del vendedor: ");
              vendedor=toupper(getche());
              putchar('\n');
              switch(vendedor)
              {
                      case 'R': printf("Ventas: $%d\n", 10000);
                                break;
                      case 'J': printf("Ventas: $%d\n", 12000);
                                break;
                      case 'M': printf("Ventas: $%d\n", 14000);
                                break;
              }


case 'C': printf("Los vendedores son Roberto, Lola y Honorario\n");
              printf("Escriba la primera letra del vendedor: ");
              vendedor=toupper(getche());
              putchar('\n');
              switch(vendedor)
              {
                      case 'R': printf("Ventas: $%d\n", 10000);
                                break ;
                      case 'L': printf("Ventas: $%d\n", 9500);
```

```
                        break ;
                case 'H': printf("Ventas: $%d\n", 13000);
                        break ;
        }

case 'O': printf("Los vendedores son Tomas, Juan y Raquel\n");
        printf("Escriba la primera letra del vendedor: ");
        vendedor=toupper(getche());
        putchar('\n');
        switch(vendedor)
        {
                case 'T': printf("Ventas: $%d\n", 5000);
                        break;
                case 'J': printf("Ventas: $%d\n", 9000);
                        break;
                case 'R': printf("Ventas: $%d\n", 14000);
                        break;
        }

   }
}
```

Vamos a ver que hace este programa:

- Observa que en este programa sólo se escoge un case por swicth(); al momento de entrar al case, el swicth() se restringe a éste y así sucesivamente hasta encontrar un break.

- Una posible salida de este programa es la siguiente:

```
(Inactive C:\TCWIN\BIN\PROG32.EXE)

Las zonas son: Este, Centro y Oeste
Escriba la primera letra de la zona:
Escriba su opcion: e
Los vendedores son Romualdo, Jeremias y Maria
Escriba la primera letra del vendedor: m
Ventas: $14000
Los vendedores son Roberto, Lola y Honorario
Escriba la primera letra del vendedor: l
Ventas: $9500
Los vendedores son Tomas, Juan y Raquel
Escriba la primera letra del vendedor: r
Ventas: $14000
```

4.7. Instrucciones de iteración

En C, como en todos los lenguajes de programación modernos, las instrucciones de iteración (también denominadas de ciclos) permiten que un conjunto de instrucciones sea ejecutado hasta que se alcance una cierta condición. Esta condición puede estar predeterminada (como en el ciclo for()) o no estar fijada de antemano (como en los ciclos while() y do-while()).

4.7.1. El ciclo for()

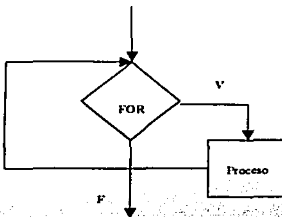
El formato general del ciclo for() de C se encuentra de una forma o de otra en todos los lenguajes de programación. C, sin embargo, proporciona una potencia y flexibilidad sorprendentes.

La forma general para la instrucción for() es:

```
for(inicialización; condición; incremento)
{
    instrucción;
}
```

El ciclo for() admite muchas variantes. Sin embargo, la inicialización normalmente es una instrucción de asignación que se utiliza para iniciar la variable de control del ciclo. La condición es una expresión relacional que determina cuándo finaliza el ciclo. El incremento define cómo cambia la variable de control cada vez que se repite el ciclo. Estas tres secciones principales deben estar separadas por punto y coma. El ciclo for() continúa ejecutándose mientras que la condición sea cierta. Una vez que la condición se hace falsa, la ejecución del programa continúa por la instrucción siguiente al for(). **El Diagrama 4.5** ilustra lo anterior.

Diagrama 4.5



- **En este diagrama podemos observar como el for() verifica la condición(s); si no se cumple sale del ciclo y si no vuelve a revisarlo hasta que no se cumpla.**

En los ciclos for(), la prueba de la condición se hace siempre al principio del ciclo. Esto supone que el código dentro del ciclo puede no ejecutarse si la condición es falsa al comienzo

4.7.1.1. Variaciones del ciclo for()

Hay variaciones del ciclo for() que permiten aumentar su potencia, flexibilidad y aplicabilidad en determinadas situaciones de programación.

Una de las variaciones más comunes utiliza el operador coma para permitir dos o más variables de control del ciclo. (Recuérdese que el operador coma se utiliza para encadenar un numero de expresiones de la forma "hacer esto y lo otro".

Las comas separan las instrucciones de inicialización. Todas éstas deben contener el valor correcto para que el ciclo for() termine.

Recuérdese que cada una de las tres secciones del ciclo for() puede consistir en cualquier expresión válida. Las expresiones no tienen incluso por qué tener que ver con el propósito general de las secciones en las que se utilizan.

Otro aspecto interesante del for() es que puede no tener todas las secciones de definición del ciclo. Esto es, no hay necesidad de se presente una expresión en cada sección; son opcionales. Por ejemplo, este ciclo se ejecuta hasta que el usuario introduce el numero 123:

```
for(x=0; x != 123; )
   scan("%d", &x);
```

Obsérvese que la parte de incremento de la definición del for() está en blanco. Esto significa que cada vez que el ciclo se repite, se analiza x para ver si es igual a 123, pero no tienen lugar acciones posteriores. Si se introduce 123 por el teclado, la condición del ciclo se hace falsa y el ciclo termina.

La inicialización de la variable de control del ciclo a menúdo se realiza fuera de la instrucción for(). La mayor parte de las veces en que así se hace, se debe a que la condición inicial de la variable de control del ciclo se debe calcular mediante alguna forma compleja, como en este ejemplo:

```
gets( c );
if ( *c)
   x=strlen ( c );
else
   x=10;
for( ; x<10 ; )
{
    printf( " %d ", x);
     ++x
}
```

- La sección de inicialización se ha dejado en blanco y x se inicializa antes de entrar en el ciclo.

4.7.1.2 El ciclo infinito

Aunque se puede utilizar cualquier instrucción de ciclo para crear un ciclo infinito, es el for() el que se usa tradicionalmente para este propósito. Como no se necesita ninguna de las tres expresiones que constituyen el ciclo for(), se puede conseguir que el ciclo no tenga fin dejando la expresión condicional vacía, como se muestra a continuacion:

```
for( ; ; )
    printf("Este ciclo estará siempre ejecutándose.\n");
```

Cuando no hay expresión condicional, se asume que es cierta. Se pueden poner expresiones de inicialización y de incremento, pero es más común entre los programadores de C utilizar la construcción for(;;) para expresar un ciclo infinito.

Realmente, la construcción for(;;) no garantiza un ciclo infinito ya que si se encuentra a la instrucción break en algún lugar dentro del cuerpo de un ciclo, da lugar a la terminación inmediata.

4.7.1.3. Los ciclos for() sin cuerpo

El cuerpo de un ciclo for() puede ser vacío. Una instrucción puede estar vacía. Esto supone que el cuerpo del ciclo for() (o de cualquier otro ciclo) puede estar vacío. El programador puede aplicar esto para aumentar la eficiencia de algunos algoritmos al igual que para originar retardos. El **Diagrama 4.6** ilustra un ciclo for() sin cuerpo.

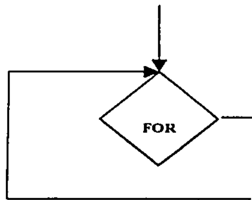


Diagrama 4.6

ESTA TESIS NO SALE DE LA BIBLIOTECA

- **En este diagrama el ciclo for() no realiza ningún proceso solo verifica la condición y se ejecuta hasta que esta sea falsa.**

Una de las tareas más comunes en programación es la eliminación de espacios en blanco de una secuencia de entrada. Por ejemplo, un programa de base de datos puede responder a la petición "mostrar los balances menores de 400". La base de datos puede necesitar tener cada una de las palabras que constituyen la petición de forma separada sin blancos. Es decir, el procesador de entrada de la base de datos puede reconocer "mostrar" pero no " mostrar". El siguiente ciclo muestra una forma de conseguir esto. Elimina los espacios en blanco iniciales que haya en la cadena apuntada por cad.

```
for( , *cad =' ' , cad++);
```

Como se puede ver, el ciclo no tiene cuerpo; ni tampoco lo necesita.

Los ciclos de retardo temporal a menúdo resultan útiles. El siguiente código muestra cómo crear uno utilizando un for():

```
for(t=0; t < ALGÚN VALOR; t++) ;
```

El único propósito de este ciclo es gastar tiempo. Sin embargo, hay que tener en cuenta que algunos compiladores optimizarán un ciclo como ése quitándolo, debido a que (por lo que al compilador respecta) ¡no tiene ningún efecto! Por tanto, puede que no siempre se consiga el retardo temporal que se pretendía.

4.7.1.4. Declaración de variables dentro de un ciclo for()

En C99 y en C++, pero no en C89, se pueden declarar variables en la sección de inicialización de un ciclo for(). Una variable declarada de esa forma tiene su ámbito limitado al bloque de código que controle esa instrucción. Es decir, una variable declarada en un ciclo for() es local a dicho ciclo.

Como normalmente la variable de control de un ciclo sólo se necesita para su ciclo, la declaración de una variable en la sección de inicialización del for() se está convirtiendo en una práctica habitual. Sin embargo, hay que recordar que esto no se puede hacer en C89.

Veamos algunos ejemplos con el ciclo for():

Programa 15

/* Este programa nos pide que introduzcamos cuantos números queremos sumar y despliega como resultado la suma de estos */

```
# include <stdio.h>

void main(void)
{
  int numero, i, suma;

  printf("\nSuma desde 0 hasta n. \n");
  printf("Introduzca el numero n: ");
  scanf("%i", &numero);
  for (i=suma=0; i <= numerous; i++)
      suma=suma + i;   /* suma+=i */
  printf("La suma desde 0 hasta %d = %d\n", numero, suma);

}
```

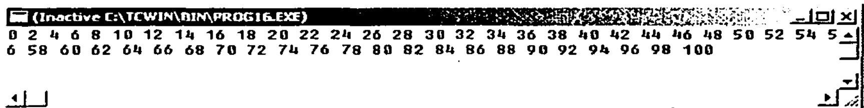
- Aquí el ciclo for() se va a ejecutar desde que i=0 hasta que i>numero; una vez que i>numero el ciclo for() termina, la variable i se va incrementando en uno cada vez que se ejecuta el for().
- En la variable suma vamos acumulando la suma de los números que vamos introduciendo y al final cuando desplegamos el resultado lo hacemos mostrando el contenido de esta variable.
- Observa la función de cada una de las partes del ciclo for(), como van cambiando mediante cada ejecución de este.
- Una salida del programa introduciendo el numero 20 es:

```
(Inactive C:\TCWIN\BIN\...
Suma desde 0 hasta n.
Introduzca el numero n: 20
La suma desde 0 hasta 20 = 210
```

Programa16

```
/* Este programa imprime en la pantalla los números pares del 0 al 100. */

# include <stdio.h>

void main(void)
{
 int x;

 for (x=0; x <= 100; x++)
 {
   if (x % 2)
    continue;    /* Regresa para incrementar y evaluar la condición */
   printf("%d ", x);
 }
}
```

Este programa despliega lo siguiente:



- ¿Por qué? La clave está en la condición del if() ya que si observamos bien la variable x empieza en cero y termina en 100, incrementándose de uno en uno. Observe que la variable x se imprime únicamente si pasa la condición del if().

Mas adelante cuando ya hayamos avanzado más en el conocimiento de C podremos hacer programas mas complejos.

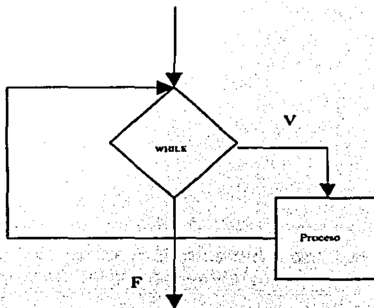
4.8. El ciclo while()

El segundo ciclo disponible en C es el ciclo while(). Su forma general es

```
while(expresión)
    {
    instrucción;
    }
```

donde instrucción es una instrucción vacía, una instrucción simple o un bloque de instrucciones. La condición puede ser cualquier expresión y cualquier valor distinto de 0 es cierto. El ciclo itera mientras la condición es cierta. Cuando la condición se hace falsa, el control del programa pasa a la línea inmediatamente siguiente al código del ciclo. El **Diagrama 4.7** ilustra lo anterior.

Diagrama 4.7



- **Observa que este diagrama es igual al 4.5, realiza prácticamente lo mismo y la única diferencia radica en los parámetros de la condición de verificación para entrar al ciclo.**

Como en el ciclo for(), el ciclo while() comprueba la condición al principio, lo que supone que el código del ciclo puede no ejecutarse si la condición es inicialmente falsa. Esto elimina la necesidad de hacer una evaluación anterior al ciclo de la condición.

No es necesario tener alguna instrucción en el cuerpo del ciclo while(). Por ejemplo:

```
while((c=getchar()) !='A') ;
```

simplemente itera hasta que se pulsa la A. Si encuentra un poco raro el poner una instrucción de asignación en la expresión condicional del while(), recuérdese que los signos iguales son simplemente operadores que devuelve el valor del operando de la parte derecha.

Programa 17

```
/* Este programa despliega los múltiplos del 4 entre 0 y 40 */

# include <stdio.h>

void main(void)
{
 int i=0, j=0, final=40;

 /* escribe los multiplos de 4 entre 0 y 40 */
 while(i < final)
 {
   i=j * 4;
   printf("%d\n", i);
   j++;  /* j=j+1 */
 }
}
```

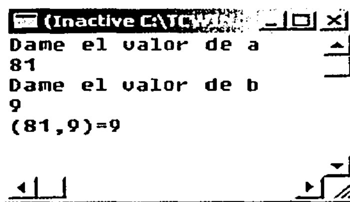
- Este programa despliega los múltiplos del 4 del 0 al 40, observa como actúa el condicional que está dentro del while().
- ¿Cómo puedo generar los múltiplos del 4 de un intervalo [0,n]? Recuerda que existe una manera para poder recibir los números del usuario y guardarlos en una variable.

Programa 18

```
/* Este programa calcula el mcd de 2 números */

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

void main()
{

int a,b,x,r,d;

clrscr();
printf ("Dame el valor de a \n");
scanf ("%d", &a);
printf ("Dame el valor de b \n");
scanf ("%d", &b);
printf ("(%d,%d)=",a,b);
```

```
 a=abs(a);
 b=abs(b);
 if (a-b>=0)
   d=a;
 else
   d=b;
 r=d;
while((a!=b)&(a!=0)&(b!=0))
{
   r=a-b;
  if(r>0)
        a=r;
   else
   {
        x=a;
        a=b;
        b=x;
   }
}
printf("%d \n",r);
getch ();
}
```

- Este programa nos pide dos números y nos muestra su mcd.
- La función abs(numero) nos da el valor absoluto del numero.
- Observa que el while() se ejecutará mientras las tres condiciones se cumplan y el cálculo del mcd se basa en un algoritmo muy sencillo que puedes ver si tomas dos números y vas siguiendo paso a paso el programa. Es un buen ejercicio hacer esto, ya que encontrarlo es un buen indicativo de que se está entendiendo que hace el programa.
- Una posible salida del programa es:



Programa 19

```
/* Este programa nos calcula algunos números que al cuadrado pueden expresarse como la suma
de otros 2 cuadrados */

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

void main(void)
{
  int x, y, z;

 printf("\n%10s %10s %10s\n", "Z", "X", "Y");
 printf("________________________________________________\n");
  x=1;
  y=1;
  while(x <= 50)
  {
     /* calcular la parte entera (z) de la raiz cuadrada */
     z=sqrt(x * y + y * y);
     while(y <= 50 && z <=50)
     {
/* Comprobar si z es suma de dos cuadrados perfectos */
          if (z * z == x * x + y * y)
   printf("\n%10d %10d %10d", z, x, y);
                  y++;  /* y=y+1 */
             z=sqrt(x * y + y * y);
     }
     x++;  /* x=x+1 */
     y=x;
  }
}
```

- En este programa es importante observar como se hace la anidación de while() (i.e. un while() dentro de otro while()) , como se le da un poquito de mas formato a la salida del programa (lo que se va a mostrar en la pantalla o se va a mandar a imprimir), la función sqrt(n) nos da la raíz cuadrada de n y para poder utilizar esta función es indispensable colocar la biblioteca math.h.

- En algunos while() se puede saber cuantos ciclos realizarán y en otros no. ¿En este programa podemos saber cuántas veces se realiza cada while()?... En uno se ejecuta el ciclo 50 veces ¿ En cual?... y ¿en el otro?

- La salida de este programa es la siguiente:

Z	X	Y
5	3	4
10	6	8
15	9	12
29	20	21

4.9. El ciclo do-while()

A diferencia de los ciclos for() y while(), que analizan la condición del ciclo al principio, el ciclo do-while() comprueba la condición al final. Esto significa que el ciclo do while() siempre se ejecuta al menos una vez. La forma general del ciclo do-while() es:

```
do
{
        instrucción;
}
while(expresión);
```

Aunque las llaves no son necesarias cuando sólo hay una instrucción, se utilizan normalmente para evitar confusiones (del lector, no del compilador) con el while().
El ciclo do-while() itera hasta que la condición se hace falsa.
El ciclo do-while() se ilustra en el **Diagrama 4.8.**

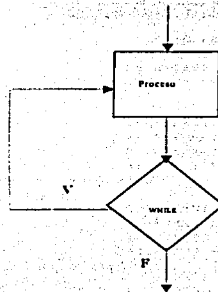
Quizá el uso más común del ciclo do-while() es en rutinas de selección por menú. Cuando el usuario introduce una respuesta válida, la devuelve como valor de la función. Una respuesta no válida hace que se muestre de nuevo el menú.

Ejemplo 20

```
/* Este programa pide un numero y mientras este sea menor que 100 se ejecuta el bloque */

# include <stdio.h>

void main(void)
{
  int num;
  do
  {
    printf("\nNumero: ");
    scanf("%d", &num);
  }
  while(num > 100);
}
```

- En este ejemplo lo único que se hace es pedir un numero mientras este sea mayor a 100.
- Observa que primero se pide el numero y después se verifica la condición (i.e. el ciclo se va a ejecutar al menos 1 vez).

Diagrama 4.8



- En este diagrama podemos ver que primero se realiza el proceso y posteriormente se verifica la condición. Si se cumple la condición vuelve a realizar el proceso de lo contrario sale del ciclo.

Ejemplo 21

```
/* Este programa se asegura que el usuario especifique la opción válida */

# include <stdio.h>

void main(void)
{
  int opción;

  /* se asegura que el usuario especifique una opción valida */
  do
  {
    printf("Transformar:\n");
    printf("\t\t 1: decimal en hexadecimal\n");
    printf("\t\t 2: hexadecimal en decimal\n");
    printf("\t\t 3: decimal en octal\n");
    printf("\t\t 4: octal en decimal\n");
    printf("Escriba su opción: ");
    scanf("%d", &opción);
  }
  while(opción < 1 || opción > 4);
}
```

- Como ya lo habíamos mencionado anteriormente el ciclo do while() ayuda a crear menús; como es el caso de este ejemplo en el cual se muestra un menú mientras no se pulse una opción correcta (i.e. el numero 1, 2, 3 ó 4) ¿Que pasa si pulsó el numero 3.6?.Una posible salida del programa es:

```
(Inactive C:\TCWIN\BIN\PROG22.EXE)
Transformar:
                        1: decimal en hexadecimal
                        2: hexadecimal en decimal
                        3: decimal en octal
                        4: octal en decimal
Escriba su opcion: 6
Transformar:
                        1: decimal en hexadecimal
                        2: hexadecimal en decimal
                        3: decimal en octal
                        4: octal en decimal
Escriba su opcion: 0
Transformar:
                        1: decimal en hexadecimal
                        2: hexadecimal en decimal
                        3: decimal en octal
                        4: octal en decimal
Escriba su opcion: 3
```

Programa 22

```
/* Este programa imprime un menú mientras el usuario asi lo desee */

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

void main(void)
{
 int opción;
 int valor;

 /* Repetir hasta que el usuario diga basta */
 do
 {
  /* asegurar que el usuario de una opción valida */
  do
  {
   printf("\ttransformar:\n");
   printf("\t\t 1: decimal en hexadecimal\n");
   printf("\t\t 2: hexadecimal en decimal\n");
   printf("\t\t 3: decimal en octal\n");
   printf("\t\t 4: octal en decimal\n");
   printf("\t\t 5: salir\n");
```

```
    printf("Escriba su opción: ");
    scanf("%d", &opción);
    putchar('\n');
   }
   while(opción < 1 || opción > 5);

   switch(opción)
   {
    case 1: printf("Introduzca un valor en decimal: ");
            scanf("%d", &valor);
            printf("%d en hexadecimal es %#x", valor, valor);
            break;
    case 2: printf("Introduzca un valor en hexadecimal: ");
            scanf("%x", &valor);
            printf("%x en decimal es %d", valor, valor);
            break;
    case 3: printf("Introduzca un valor en decimal: ");
            scanf("%d", &valor);
            printf("%d en octal es %#o", valor, valor);
            break;
    case 4: printf("Introduzca un valor en octal: ");
            scanf("%o", &valor);
            printf("%o en decimal es %d", valor, valor);
            break;
   }
   putchar('\n');
  }
  while(opción!=5);
}
```

- En este programa ya podemos apreciar un poquito más algo de lo que hemos aprendido en este capítulo; de hecho este programa está construido juntando ordenadamente dos anteriores y un do- while(). ¿Puedes decir que hace? ¿Cómo? y ¿Por qué?

4.10. Instrucciones de salto

C tiene cuatro instrucciones que llevan a cabo un salto incondicional: return, go to, break y continue. De ellas se pueden usar return y go to en cualquier parte dentro de una función y las instrucciones break y continue se usan junto con instrucciones de ciclo. Como ya se explicó anteriormente, también se puede usar break con switch().

4.10.1. La instrucción return

La instrucción return se usa para volver de una función. Se trata de una instrucción de salto porque hace que la ejecución vuelva (salte atrás) al punto en que se hizo

la llamada a la función. Un return puede tener o no un valor asociado. Sólo se puede utilizar un return con un valor asociado en una función con valor de vuelta de tipo distinto de void. En ese caso, el valor asociado con return es el valor de vuelta de la función. Un return sin valor de vuelta se utiliza para volver de las funciones de tipo void.

Técnicamente, en C89 no es necesario que se devuelva un valor con return en las funciones de tipo distinto de void. Si no se especifica un valor, se devuelve un valor sin sentido. Sin embargo, en C99 return debe devolver explícitamente un valor en las funciones que no sean de tipo void. (Esto también se aplica a C++). Por supuesto, incluso en C89, si se declara una función como devolviendo un valor, lo mas correcto es devolver realmente uno.

La forma General de la instrucción return es

```
return expresión;
```

La expresión estará presente sólo si se ha declarado la función como devolviendo un valor. En este caso, el valor de la expresión se convertirá en el valor devuelto por la función.

Se pueden usar tantas instrucciones return como se quiera en una función. Sin embargo, la función termina tan pronto como se encuentra el primer return. La llave } que termina el código de la función también hace que se vuelva de ella. Equivale a un return sin valor específico. Si esto ocurre en una función declarada de un tipo distinto del void, entonces el valor de vuelta de la función queda indeterminado.

Una función declarada como void no puede contener una instrucción return que especifique un valor. Dado que las funciones void no devuelven valor, parece lógico que no contengan ninguna instrucción return que devuelva un valor.

3.10.2. La instrucción go to

Como C tiene un rico conjunto de estructuras de control y permite un control adicional usando break y continue, no es muy necesario el go to. La idea principal que los programadores tienen sobre el go to es que tiende a hacer los programas ilegibles. De todas formas, aunque el uso del go to ha decaído desde hace algunos años, ocasionalmente tiene su utilidad. Aunque no hay situaciones de programación que necesiten el uso del go to, resulta conveniente y puede ser beneficioso, si se usa apropiadamente, en ciertas situaciones de programación, como salir de un grupo de ciclos muy anidados.

La instrucción go to necesita una etiqueta para operar. (Una etiqueta es un identificador válido seguido por dos puntos.) Además, la etiqueta debe estar en la misma función en la que se utiliza el go to; no se puede saltar entre funciones. La forma general de la instrucción go to es:

```
go to etiqueta;
etiqueta:
```

donde etiqueta es cualquier etiqueta válida anterior o posterior al go to. Por ejemplo, se podría escribir un ciclo desde 1 hasta 100 utilizando un go to y una etiqueta, como se muestra a continuación:

```
x = 1;
ciclo1:
if(x <= 10)
    go to ciclo1;
```

4.10.3. La instrucción break

La instrucción break tiene dos usos. Se puede usar para finalizar un case en una instrucción switch() y para forzar la terminación inmediata de un ciclo, saltando la evaluación condicional normal del ciclo.

Cuando se encuentra la instrucción break dentro de un ciclo, el ciclo finaliza inmediatamente y el control pasa a la instrucción que sigue al ciclo. Por ejemplo:

Programa 23

```
/* En este programa se muestra como se interrumpe un ciclo utilizando la instrucción break */

#include <stdio.h>
int main(void)
{
int t;
for (t=0; t < 100; t++)
{
   printf("%d ', t);
   if (t == 10)
  break;
}
return 0;
}
```

- Este programa imprime los números del 0 al 10 en la pantalla. A continuación el ciclo termina ya que la instrucción break da lugar a la salida inmediata del ciclo, pasando por alto la prueba condicional t<100.

La instrucción break principalmente es utilizada por los programadores en ciclos donde una condición especial debe provocar la terminación inmediata.

Un break da lugar a la salida sólo del ciclo más interno. Por ejemplo,

```
for(t=0; t < 100; ++t)
{

        cont = 1;
        for(;;)
        {
        printf("%d , cont);
        cont++;
        if(cont==10)
        break;

        }
}
```

- Este programa imprime los números del 1 al 10 en la pantalla cien veces. Cada vez que el compilador encuentre break, el control vuelve al ciclo for() externo.

Obs. Un break utilizado en una instrucción switch() afecta sólo a ese switch(). No afecta a ningún ciclo en el que se encuentre el switch().

4.10.4. La función exit()

Aunque exit() no es una instrucción de control de programa, procede explicarla aquí ya que la usaremos en algunos programas posteriores. Igual que se puede interrumpir un ciclo, se puede salir anticipadamente de un programa usando la función exit() de la biblioteca estándar. Esta función da lugar a la terminación inmediata del programa, forzando la vuelta al sistema operativo. El efecto de exit() es el de un break que afecta al programa entero. Veamos un ejemplo:

Programa 24

```
/* Este programa ilustra la salida de un programa mediante el uso de la instrucción exit ( ) */

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

void main(void)
{
char c;

printf("Revisar Calificaciones\n");
printf("1. Calculo \n");
printf("2. Álgebra \n");
printf("3. Geometria \n");
printf("4. Salir \n");
printf("Introduzca su opción:  ");
do
{
 c=getchar();
 switch(c)
 {

   case '1':
     printf("Tu calificación de Calculo es : %d",10);
     break;
   case '2':
     printf("Tu calificación de Álgebra es : %d",10);
     break;
   case '3':
     printf("Tu calificación de Geometria es : %s","NA");
     break;
   case '4':
     exit(0);

 }
}
while (c!='1' && c!=2 && c!=3);
}
```

- Notemos que al pulsar la opción 4 el programa automáticamente termina. ¿Se puede salir del programa de otra manera? Si no ¿Qué le modificarías para que se pudiera salir?

4.10.5. La instrucción continue

La instrucción continue funciona de una forma algo similar a break. Sin embargo, en vez de forzar la terminación, continue fuerza una nueva iteración del ciclo y salta cualquier código que exista entre medias. Para el ciclo for(), continue hace que se ejecuten los planes de prueba condicional y de incremento del ciclo. Para los ciclos while() y do-while(), el control del programa pasa a la prueba condicional.

Ejemplo 25

```
/* Este programa cuenta los espacios en blanco de una cadena proporcionada por el usuario */

#include <stdio.h>

int main (void)
{
char c[80], *cad;
int espacios;

printf("Introduzca una cadena: ");
gets(c) ;
cad = c ;
for(espacios=0; *cad; cad++)
{
if(*cad!=' ')
  continue ;
espacios++ ;
  }
 }
printf("%d espacios\n", espacios);
return 0;
}
```

- Este programa cuenta los espacios en blanco de una cadena proporcionada por el usuario.
- Se comprueba cada carácter para ver si es un espacio. Si no lo es, la instrucción continue fuerza una nueva iteración del for(). Si el carácter es un espacio, se incrementa espacios.
- Una posible salida de este programa es :

```
(Inactive C:\TCWIN\BIN\PROG22Z.EXE)
Introduzca una cadena: Programacion en C
2 espacios
```

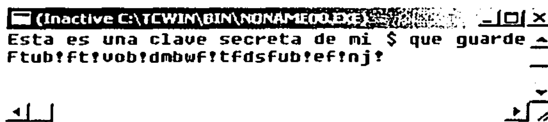
El ejemplo siguiente muestra cómo se puede utilizar continue para acelerar la salida del ciclo forzando la evaluación de la condición antes de lo normal:

Programa 26

```
/* Este programa muestra el uso de continue con el ciclo while */

# include <stdio.h>
# include <conio.h>
void main(void)
{

char hecho, c ;
hecho = 0;

while(!hecho)
{
  c = getchar() ;
  if (c = = '$')
  {
  hecho = 1;
  continue;
  }
  putchar(c+1) ;
}
}
```

- Esta función codifica un mensaje desplazando cada letra a la siguiente del alfabeto. Por ejemplo, la letra A se transformaría en una B. La función finalizaría cuando el programa leyera un $. Una vez leído $ no se producirán posteriores salidas ya que se forzó la evaluación de la condición por efecto del continue, que evaluaría hecho a cierto y daría lugar a la salida del ciclo.

- Una salida de codificación de código con este programa es:



Capítulo 5
Datos Estructurados

5.1. Arreglos y Cadenas

Un arreglo es una colección de variables del mismo tipo que se referencian por un nombre común. A un elemento específico de un arreglo se accede mediante un índice. En C, todos los arreglos constan de posiciones de memoria contiguas. Los arreglos pueden tener de una a varias dimensiones. El arreglo más común es la cadena, que simplemente es un arreglo de caracteres terminado por un nulo.

Los arreglos y los apuntadores están íntimamente relacionados; pero por el momento nos centraremos en los arreglos.

5.1.1. Arreglos unidimensionales

La forma general de declaración de un arreglo unidimensional es

```
tipo nombre de variable[tamaño];
```

Como en otros lenguajes, los arreglos tienen que declararse explícitamente para que así el compilador pueda reservar espacio en memoria para ellos. Aquí tipo indica el tipo base del arreglo, que es el tipo de cada elemento del arreglo. El valor de tamaño indica cuántos elementos mantendrá el arreglo. Por ejemplo, para declarar un arreglo de 100 elementos de tipo double denominado balance se usa la instrucción:

```
double balance[100];
```

En C89, el tamaño de un arreglo se debe especificar por medio de una expresión constante. Así, en C89 el tamaño de los arreglos se fija en tiempo de compilación. (C99 permite declarar arreglos cuyo tamaño se establece en tiempo de ejecución).

Un elemento se puede acceder indexando el nombre del arreglo. Esto se hace colocando el índice del elemento entre corchetes justo detrás del nombre del arreglo. Por ejemplo:

```
balance[3] = 12.23;
```

asigna al elemento número 3 de balance el valor 12.23. En C, todos los arreglos tienen el 0 como índice de su primer elemento. Por tanto, cuando se escribe

```
char p[10];
```

se está declarando un arreglo de caracteres que tiene diez elementos, desde p[0] hasta p[9].

C no comprueba los límites de los arreglos. Se puede sobrepasar cualquier extremo de un arreglo y escribir en los datos de algunas otras variables o incluso en el código del programa. Como programador, es tarea suya proporcionar una comprobación de límites cuando sea necesario. Por ejemplo, este código compilará sin errores, pero es incorrecto porque el ciclo for hace que se sobrepase el final del arreglo cont.

```
int cont[10], i;

/* esto hace que se sobrepase el final de cont */

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

5.1.2. Generación de un apuntador a un arreglo

En C el nombre de un arreglo representa la dirección constante de su primer elemento. Por ejemplo, dado

```
int contador[10];
```

se puede acceder al primer elemento usando simplemente el nombre contador; por ejemplo, el siguiente fragmento de programa asigna a p la dirección del primer elemento de contador.

```
int *p;
int contador[10];
p = contador;
```

También se puede especificar la dirección del primer elemento de un arreglo utilizando el operador &. Por eso, contador y &contador[0] producen ambos el mismo resultado. Sin embargo, en código de C escrito de forma profesional casi nunca se vería &contador[0].

5.1.3. Paso de arreglos unidimensionales a funciones

En C no se puede pasar un arreglo completo como argumento a una función. Sin embargo, se puede pasar a una función un apuntador a un arreglo especificando el nombre del arreglo sin índice. Por ejemplo, el siguiente fragmento de programa pasa la dirección de i a funcl().

```
int main(void)
{
int i[10];
funcl(i);
/*...*/
```

Si una función recibe un arreglo unidimensional, se puede declarar el parámetro formal de tres formas: como un apuntador, como un arreglo delimitado o como un arreglo no delimitado. Por ejemplo, para recibir x en una función llamada funcl(), se puede declarar funcl() como:

Un apuntador.

```
void funcl(int *x)
```

Un arreglo delimitado.

```
void funcl(int x[10]).
```

Un arreglo no delimitado.

```
void funcl(int x[])
```

El resultado de las tres formas de declaración es idéntico porque cada uno le indica al compilador que se va a recibir un apuntador a entero. En la primera declaración se usa realmente un apuntador. En la segunda se emplea la declaración estándar de arreglo. En la última declaración, la versión modificada de la declaración del arreglo simplemente especifica que se va a recibir un arreglo de tipo int de cierta longitud. Como se puede ver, por lo que a la función respecta, realmente no importa la longitud del arreglo porque C no comprueba los límites. De hecho, por lo que al compilador se refiere,

```
void funcl(int x[32])
```

también sirve, porque el compilador de C genera código que avisa a funcl() que va a recibir un apuntador; realmente no crea un arreglo de 32 elementos.

5.1.4. Cadenas

El uso más común de los arreglos unidimensionales; es como cadenas de caracteres. En C, una cadena es un arreglo de caracteres terminado en nulo. (Un nulo es un cero). Por tanto, una cadena contiene los caracteres que la conforman seguidos de un nulo. La cadena terminada en nulo es el único tipo de cadena definido en C.

Cuando se declara un arreglo de caracteres para una cadena, es necesario que se declare como de un carácter más que la cadena más larga que pueda contener. Por ejemplo, para declarar un arreglo cad que contenga una cadena de hasta 10 caracteres, se escribirá

```
char cad[11];
```

Esto dejará sitio para el carácter nulo del final de la cadena. Cuando en un programa se utiliza una constante de cadena entre comillas, realmente se está creando una cadena terminada en nulo. Una *constante de cadena* es una lista de caracteres encerrada entre comillas dobles; por ejemplo:

"hola amigos"

No es necesario añadir explícitamente el carácter nulo al final de las constantes de cadena; el compilador lo hace automáticamente. C incluye una gran variedad de funciones de manipulación de cadenas. Las más comunes son las siguientes:

Nombre	Función
strcpy(cl, c2)	Copia c2 en cl.
strcat(cl, c2)	Concatena c2 al final de cl.
strlen(cl)	Devuelve la longitud de cl.
strcmp(cl, c2)	Devuelve 0 si cl y c2 son iguales; menor que 0 si cl <c2; mayor que 0 si c1 >cl.
strchr(cl, car)	Devuelve un apuntador a la primera ocurrencia de car en cl.
strstr(cl, c2)	Devuelve un apuntador a la primera ocurrencia de c2 en cl.

Estas funciones usan el archivo de cabecera estándar <string.h>.

Ejemplos:

Programa 27

```
/* Este programa copia una cadena a una variable */

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

void main(void)
{
  char saludo[80];

  /* strcpy copia la cadena "hola" a la variable saludo */
  strcpy(saludo, "hola");
  printf("%s", saludo);
}
```

- Podemos ver lo que hace strcpy() en este programa como saludo ='hola'

Programa 28

```
/* Este programa concatena el contenido de 2 cadenas */

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

void main(void)
{
  char saludo1[20], saludo2[20];

  strcpy(saludo1, "hola");
  strcpy(saludo2, " amigos");

  strcat(saludo1, saludo2);
  printf("%s", saludo1);
}
```

- Este programa nos muestra "hola amigos".
- strcat() concatena el contenido de saludo1 y saludo2 y lo almacena en saludo1.

Programa 29

```
/* Este programa compara dos cadenas y nos regresa el valor de la comparación */

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

void main(void)
{
  char contraseña[80];

  printf("\n Introduzca contraseña: ");
  gets(contraseña);

  if (strcmp(contraseña, "curso"))
    printf("Contraseña incorrecta\n");
  else
    printf("Contraseña correcta\n");
}
```

- Este programa nos pide una cadena y la compara con la cadena 'curso'
- strcmp() compara dos cadenas y nos regresa un valor de la comparación, de la siguiente manera:

  0 si son iguales
  < 0 si la primera es menor que la segunda
  0 si la primera es mayor que la segunda

- Recuerde que strcmp() devuelve falso si las cadenas son iguales. Por tanto, si se está probando la igualdad, hay que asegurarse de usar el operador lógico ! para invertir la condición.

Programa 30

```
/* En este programa, dada una cadena nos dice cuantos caracteres tiene. */

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

void main(void)
{
  char cadena[80];

  printf("\n Introduzca una cadena: ");
  gets(cadena);

    printf("La cadena tiene una longitud de %d caracteres", strlen(cadena));
}
```

- Este programa nos pide una cadena y nos regresa cuantos caracteres tiene.
- strlen() nos regresa la longitud de una cadena.
- ¿Puedo escribir espacios en blanco? ¿Qué pasa si me paso de 80 caracteres?

5.1.5. Arreglos bidimensionales

C admite arreglos multidimensionales. La forma más simple de un arreglo multidimensional es el arreglo bidimensional. Un arreglo bidimensional es esencialmente un arreglo de arreglos unidimensionales. Para declarar un arreglo "d" de enteros bidimensional de tamaño 5,10 se escribirá

```
int d[5][10];
```

Préstese atención a la declaración. Muchos otros lenguajes de computadora utilizan comas para separar las dimensiones del arreglo; C coloca cada dimensión en su propio conjunto de corchetes.

De forma similar, para acceder al punto (1,2) del arreglo d se escribirá

    d[1][2]

La **Figura 5.1** ilustra el arreglo d[5][10].

Figura 5.1

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										

Programa 31

```
/* En este programa llena un array y nos despliega su contenido */

# include <stdio.h>

void main(void)
{
  int t, i, num[3][4];
  clrscr();
  for (t=0; t<3; ++t)
  {
    for (i=0; i<4; ++i)
         num[t][i]=(t*4)+i+1;
  }

  /* Ahora imprimirlos */
  for (t=0; t<3; ++t)
  {
    for (i=0; i<4; ++i)
         printf("%3d",num[t][i]);
    printf("\n");
  }
}
```

- Este programa llena el arreglo bidimensional num con los números del 1 al 12 y nos despliega su contenido.

Los arreglos bidimensionales se almacenan en matrices fila-columna, en las que el primer índice indica la fila y el segundo indica la columna.

Cuando se utiliza un arreglo bidimensional como argumento de una función, realmente sólo se pasa un apuntador al primer elemento. Sin embargo, la función que recibe un arreglo bidimensional como parámetro tiene que definir al menos la longitud de la segunda dimensión. (Se puede establecer también la primera dimensión si se quiere, pero no es necesario). La segunda dimensión es necesaria porque el compilador necesita saber la longitud de cada fila para indexar el arreglo correctamente. Por ejemplo, una función que recibe un arreglo entero bidimensional de dimensiones 10,10 se declara así:

```
void funcl(int x[][10])
/* ... */
```

El compilador necesita conocer la segunda dimensión para poder trabajar con declaraciones como:

```
x[2] [4]
```

dentro de la función. Si la longitud de las filas está sin definir, entonces sería imposible saber dónde empieza la tercera fila.

5.1.6. Arreglos y cadenas

Para ejemplificar un arreglo de cadenas veamos el siguiente ejemplo:

Programa 32

```
/* Este programa crea una base de datos de 10 cadenas */

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

# define MAX 10    /* máximo número de nombres */
# define LONG 60   /* número de caracteres máximo por nombre */

void main(void)
```

```
{
  char lista[MAX][LONG];
  int i=-1, n; /* indices */

  clrscr();
  puts("Para finalizar oprime Enter");
  do
  {
    printf("\n Nombre y Apellidos : ");
    gets( lista[++i]);
  }
  while (strlen(lista[i]) != 0 && (i+1) < MAX);

  /* Escribir datos */

  printf(" \n\n");
  for (n=0; n<=i; n++)
    printf("%s \n", lista[n]);

}
```

- Este programa crea una pequeña base de datos con 10 cadenas, con un máximo de 80 caracteres cada una. Cada una de las cadenas las introducimos conforme no las va pidiendo el mismo programa.

Observemos que "define MAX 10" crea e inicializa la constante simbólica MAX con 10, i.e. crea una constante la cual no se podrá modificar a lo largo del programa. Esto es muy útil porque si quisiéramos que la base de datos fuera de n cadenas bastaría con corregir esa línea colocando en lugar de 10 un número n; con esto automáticamente cada una de las líneas donde se utiliza la constante se actualiza.

5.1.7. Arreglos de cadenas

No es raro en programación utilizar un arreglo de cadenas. Por ejemplo, el procesador de entrada a una base de datos puede verificar las órdenes del usuario comparando con un arreglo de órdenes válidas. Para crear un arreglo de cadenas se utiliza un arreglo de caracteres bidimensional. El tamaño del índice izquierdo determina el número de cadenas y el tamaño del índice derecho especifica la longitud máxima de las cadenas. El código que sigue declara un arreglo de 30 cadenas, cada una con una longitud máxima de 79 caracteres.

```
char arreglo_cad[30][80];
```

Es bastante fácil acceder a una cadena individual: simplemente se especifica sólo el índice izquierdo. Por ejemplo, la siguiente declaración llama a gets() con la tercera cadena de arreglo_cad.

```
gets(arreglo_cad[2]);
```

La declaración anterior equivale funcionalmente a

```
gets(&arreglo_cad[2][0]);
```

pero la primera de las dos formas es mucho más común en el código en C profesional.

5.1.8. Arreglos multidimensionales

C permite arreglos de más de dos dimensiones. La forma general de declaración de un arreglo multidimensional es:

```
tipo nombre(Tamañol][Tamaño2][Tamaño3]...[TamañoN];
```

Los arreglos de tres o más dimensiones no se utilizan a menudo por la cantidad de memoria que se requiere para almacenarlos. Con los arreglos multidimensionales el cálculo de cada índice le lleva a la computadora una significativa cantidad de tiempo. Esto significa que el acceso a un elemento en un arreglo multidimensional tarda más que el acceso a un elemento en un arreglo unidimensional.

Cuando se pasan arreglos multidimensionales a funciones, se tienen que declarar todas excepto la primera dimensión. Por ejemplo, si se declara un arreglo m como:

```
int  m[4] [3] [6] [5] ;
```

una función funcl() que reciba a m podría ser como:

```
void  funcl (int d[][3][6][5])
          /* .... */
```

Por supuesto, si se quiere se puede incluir la primera dimensión.

5.1.9. Indexación de apuntadores

Los apuntadores y los arreglos están estrechamente relacionados. Como ya sabemos, un nombre de arreglo sin índice es un apuntador al primer elemento del arreglo. Por ejemplo, considérese el siguiente arreglo.

```
char p[10];
```

Las declaraciones que siguen son idénticas:

```
p, &p[0]
```

Puesto de otro modo,

```
p = = &p[0]
```

se evalúa igual porque la dirección del primer elemento del arreglo es la misma que la dirección del arreglo.

Como ya se ha indicado, un nombre de un arreglo representa un apuntador constante al primer elemento. Por otro lado, cualquier variable apuntador puede indexarse como si estuviera declarada como un arreglo. Por ejemplo, considérese este fragmento de programa:

```
int *p, i[10];
p = i;
p[5] = 100; /* asignación usando índice */
*(p+5) = 100; /* asignación usando aritmética de apuntadores */
```

Ambas instrucciones de asignación ponen el valor 100 en el sexto elemento de i. La primera instrucción indexa p; la segunda utiliza la aritmética de apuntadores. De ambas maneras el resultado es el mismo.

Este mismo concepto se puede aplicar también a los arreglos de dos o más dimensiones.. En general, para cualquier arreglo bidimensional:

```
a[j][k] es equivalente a *((tipo base *)a+(j * longitud de la fila) + k)
```

El molde que convierte de apuntador a arreglo a apuntador al tipo base es necesario para que funcione correctamente la aritmética de apuntadores . A veces se utilizan los apuntadores para acceder a los arreglos debido a que la aritmética de apuntadores es normalmente más rápida que la indexación de arreglos.

5.1.10. Inicialización de arreglos

C permite la inicialización de arreglos en el momento de declararlos. La forma general de inicialización de un arreglo, que aparece a continuación, es similar a la de otras variables:

```
tipo nombre de arreglo [tamañol]...[tamañoN] = {lista_de_valores};
```

La lista_de_valores es una lista de constantes separadas por comas cuyo tipo es compatible con el especificador de tipo. La primera constante se coloca en la

primera posición del arreglo, la segunda constante en la segunda posición, y así sucesivamente. Obsérvese que un punto y coma sigue a }.

En el ejemplo que sigue se inicializa un arreglo de enteros de 10 elementos con los números del 1 al 10.

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, l0};
```

Esto significa que i[0] tiene el valor 1 e i[9] el valor 10.
Los arreglos de caracteres que contienen cadenas permiten una inicialización abreviada de la forma:

```
char nombre_de arreglo(tamañoJ = "cadena ");
```

Por ejemplo, este fragmento de código inicializa cad con la frase "Me gusta C".

```
char cad[11] = "Me gusta C";
```

El código anterior produce el mismo resultado que si se escribe

```
char cad[11] _ {'M', 'e', ' ', 'g', 'u', 's', 't', 'a', ' ', 'C", '\0'};
```

Debido a que todas las cadenas en C terminan con un nulo, se debe estar seguro de que el arreglo que se declara es suficientemente largo para incluirlo. Esto es por lo que cad es de once caracteres de longitud aunque "Me gusta C" tiene sólo diez caracteres. Cuando se utiliza una cadena constante, el compilador automáticamente proporciona la terminación nula.

Los arreglos multidimensionales se inicializan del mismo modo que los unidimensionales. Por ejemplo, lo siguiente inicializa cuads con los números del 1 al 10 y sus cuadrados.

```
int cuads[10][2]  {
        1, 1,
        2, 4,
        3, 9,
        4, 16,
        5, 25,
        6, 36,
        7, 49,
        8, 64,
        9, 81,
        10, 100
        };
```

Cuando se inicializan arreglos multidimensionales se pueden añadir llaves para encerrar los inicializadores de cada dimensión. Esto se denomina agrupamiento

subagregado. Por ejemplo, la declaración anterior se podría haber escrito también como:

```
int cuads[10][2] {
        {1,1},
        {2,4},
        {3,9},
        {4, 16},
        {5, 25},
        {6, 36},
        {7, 49},
        {8 64},
        {9, 81},
        {10, 100);
};
```

Obs. Cuando se utiliza agrupamiento subagregado, si no se proporcionan suficientes inicializadores para un grupo dado, los restantes miembros se inicializan a cero automáticamente.

5.1.11. Inicialización de arreglos no delimitados

Imaginemos que se está utilizando una inicialización de arreglo para construir una tabla de mensajes de error como la que se muestra aquí:

```
char e1[18] = "Error de lectura\n";
char e2[20] = "Error de escritura\n";
char e3[30] = "No se puede abrir el archivo\n";
```

Como se podrá suponer, resulta tedioso contar los caracteres de cada mensaje manualmente para determinar la dimensión correcta del arreglo. Se puede hacer que el compilador calcule automáticamente las dimensiones de los arreglos. Si en una instrucción de inicialización de un arreglo no se especifica el tamaño del arreglo, el compilador automáticamente crea un arreglo suficientemente grande para mantener todos los inicializadores presentes. Esto es lo que se denomina arreglo no delimitado. Utilizando este método, la tabla de mensajes queda como:

```
char el[] = "Error de lectura\n";
char e2[] = "Error de escritura\n";
char e3[] _ "No se puede abrir el archivo\n";
```

Además de ser menos tedioso, el uso de la inicialización de arreglos no delimitados permite cambiar cualquiera de los mensajes sin temor a teclear una cuenta incorrecta accidentalmente.

El uso de inicializaciones de arreglos no delimitados no está restringido a los arreglos unidimensionales. Para los arreglos multidimensionales se deben

especificar todas excepto la dimensión más a la izquierda. (Las otras dimensiones se necesitan para permitir al compilador de C indexar el arreglo adecuadamente.) De este modo se puede construir tablas de longitudes variables, y el compilador asignará automáticamente suficiente espacio para ellas. Por ejemplo, a continuación se muestra la inicialización del arreglo no delimitado.

```
int cuads[ ][2] = {
        {1, 1},
        {2, 4},
        {3, 9},
        {4, 16},
        {5, 25},
        {6, 36},
        {7, 49},
        };
```

La ventaja de esta declaración sobre la versión de tamaño delimitado es que la tabla puede alargarse o acortarse sin tener que cambiar las dimensiones del arreglo.

5.2. Apuntadores

El correcto entendimiento y uso de los apuntadores es crítico para una fructífera programación en C. Los apuntadores son una de las características más poderosas de C, pero también una de las más peligrosas. Por ejemplo, un apuntador que no contenga un valor válido puede provocar el fallo del sistema. Y lo que quizá es peor, es fácil utilizar apuntadores de forma incorrecta, y el uso incorrecto puede causar fallos muy difíciles de encontrar.

En este trabajo se darán únicamente los elementos básicos para trabajar con apuntadores; el lector interesado en profundizar mas en el uso de estos podrá encontrar mas información en la bibliografía.

5.2.1 ¿Qué son los apuntadores ?

Un apuntador es una variable que contiene una dirección de memoria. Por ejemplo, si una variable contiene la dirección de otra variable, entonces se dice que la primera variable apunta a la segunda.

5.2.2 Variables apuntador

Si una variable va a ser un apuntador, entonces tiene que declararse como tal. Una declaración de apuntador consiste en un tipo base, un * y el nombre de la variable. La forma general de declaración de una variable apuntador es:

```
tipo * nombre;
```

donde tipo es el tipo base del apuntador, que puede ser cualquier tipo válido. El nombre de la variable apuntador se especifica con nombre.

El tipo base del apuntador define el tipo de variables a las que puede apuntar el apuntador. Técnicamente, cualquier tipo de apuntador puede apuntar a cualquier lugar de la memoria. Sin embargo, todas las operaciones de apuntadores se realizan de acuerdo con sus tipos base. Por ejemplo, cuando se declara un apuntador como de tipo int *, el compilador asume que cualquier dirección que contenga apunta a un entero; lo haga realmente o no. (i. e., un apuntador int * siempre "piensa" que apunta a un objeto int, independientemente de lo que realmente contenga esa parte de la memoria). Por tanto, al declarar un apuntador hay que asegurarse de que su tipo base sea compatible con el tipo base al que se quiera que apunte.

5.2.3. Los operadores de apuntadores

Ya se comentaron los operadores de apuntadores anteriormente. Aquí los repasaremos. Hay dos operadores de apuntadores : & y *. & es un operador unario que devuelve la dirección de memoria de su operando. (Recuérdese que un operador unario sólo necesita un operando.) Por ejemplo,

```
m = &cuenta;
```

pone en m la dirección de memoria de la variable cuenta. Esta dirección es la posición interna de la variable en la computadora. No tiene nada que ver con el valor de cuenta. Se puede pensar en el operador & como devolviendo "la dirección de". Por tanto, la instrucción de asignación anterior significa "m recibe la dirección de cuenta".

Para entender mejor la asignación anterior, supongamos que la variable cuenta utiliza la posición de memoria 2000 para guardar su valor. Supongamos también que cuenta contiene el valor 100. Entonces, después de la asignación anterior, m contiene el valor 2000.

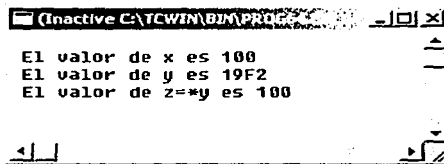
El segundo operador de apuntadores "*" es el complemento de &. Es un operador unario que devuelve el valor que se encuentra en la dirección que le sigue. Por ejemplo, si m contiene la dirección de memoria de la variable cuenta, entonces pone el valor de cuenta en q. Así, q tendrá el valor 100, porque 100 es lo almacenado en la posición 2000, la dirección de memoria que se guardó en m. Se puede pensar en * como "en la dirección". En este caso, la instrucción anterior significa "q recibe el valor en la dirección m". Ejemplos:

Programa 33

/* Este programa nos muestra el uso de los operadores puntero y los modificadores que deben de utilizarse con printf para generar la salida correcta */

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

int x;
int *y;
int z;

clrscr();
x=100;
y=&x;
z=*y;
printf("\n El valor de x es %d", x);
printf("\n El valor de y es %p",y);
printf("\n El valor de z=*y es %d", z);

}
```

- Este programa nos da la siguiente salida:



- Observemos el uso de los operadores de apuntadores y de cada uno de los modificadores del printf que se usan para desplegar el valor de cada variable.

Programa 34

```
/* Este programa ejemplifica el uso correcto de punteros */

# include<stdio.h>

void main(void)
{

  float n1=3.54, n2, *apuntador1, *apuntador2, auxiliar;

  apuntador2=&n1;
  n2 = *apuntador2;
  apuntador1=&auxiliar;
  *apuntador1=20.0;
  printf("En la dirección %p esta  el dato %f \n", apuntador2, n2);
  printf("En la dirección %p esta  el dato %f \n", apuntador1, *apuntador1);

}
```

Este programa nos da la siguiente salida.

```
(Inactive C:\TCWIN\BIN\PROG34.EXE)
En la dirección 1BDA esta  el dato 3.540000
En la dirección 1BD2 esta  el dato 20.000000
```

Donde podemos observar que:

- " apuntador2 =&n1 " asigna la dirección de n1 a la variable apuntador2
- " n2 = *apuntador2 " asigna a n2 el contenido de la dirección apuntador2

5.2.4. Expresiones de apuntadores

En general, las expresiones que involucran apuntadores siguen las mismas reglas que las demás expresiones. En esta sección se examinarán algunos aspectos especiales de las expresiones de apuntadores , tales como las asignaciones y aritmética de estos.

5.2.5. Asignaciones de apuntadores

Un apuntador puede utilizarse a la derecha de una instrucción de asignación para asignar su valor a otro apuntador. Cuando ambos apuntadores son del mismo tipo, la situación es muy sencilla. Por ejemplo:

Programa 35

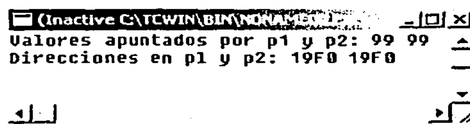
```
/* Este programa ejemplifica el uso de asignaciones con punteros */

#include <stdio.h>

int main (void)
{
int x = 99;
int *p1, *p2;

p1 = &x;
p2 = p1;
printf("Valores apuntados por p1 y p2: %d %d\n",*p1, *p2);
printf("Direcciones en p1 y p2: %p %p", p1, p2);
return 0;
}
```

- Tras la secuencia de asignaciones

```
p1 = &x;
p2 = p1;
```

  p1 y p2 apuntan a x. Por consiguiente, p1 y p2 se refieren al mismo objeto.

- A continuación se muestra un ejemplo de salida del programa, confirmando esto.



- Donde podemos observar que las direcciones se muestran usando el modificador de formato de printf() %p, que hace que printf() muestre una dirección en el formato utilizado por la computadora en cuestión.

También se puede asignar a un apuntador de un tipo otro apuntador de otro tipo. Sin embargo, eso involucra una conversión de apuntadores; la cual, no trataremos en este trabajo.

5.2.6. Aritmética de apuntadores

Existen sólo dos operaciones aritméticas que se pueden usar con apuntadores : la suma y la resta. Para entender qué ocurre en la aritmética de apuntadores , sea p1 un apuntador a un entero con valor actual de 2000. Además, asumamos que los enteros son de dos bytes de longitud. Después de la expresión

```
p1++;
```

p1 contiene 2002, no 2001. La razón de esto es que cada vez que p1 se incrementa, apunta al siguiente entero. Lo mismo ocurre al decrementar. Por ejemplo, suponiendo que p1 tiene el valor 2000, la expresión

```
p1--;
```

hace que p1 tenga el valor 1998.

Generalizando a partir del ejemplo anterior, las siguientes son las reglas que gobiernan la aritmética de apuntadores. Cada vez que se incrementa en uno un apuntador, apunta a la posición de memoria del siguiente elemento de su tipo base. Cada vez que se decrementa en uno, apunta a la posición del elemento anterior. Cuando se aplica un incremento o decremento a apuntadores de tipo char, esto hace que parezca una aritmética "normal", ya que los caracteres siempre ocupan un byte. El resto de los apuntadores aumentan o decrecen en la longitud del tipo de datos a los que apuntan. Este método asegura que un apuntador siempre apunta a un elemento adecuado del tipo base.

La aritmética de apuntadores no está limitada sólo a los operadores de incremento y decremento. Por ejemplo, se pueden sumar o restar enteros a y de apuntadores . La expresión

```
p1 = p1 + 12;
```

hace que p1 apunte al duodécimo elemento del tipo de p1 que está más allá del elemento al que apunta actualmente.

Además de la suma y la resta de un apuntador y un entero, sólo se puede realizar otra operación aritmética más: restar un apuntador a otro con el fin de obtener el número de objetos del tipo base que separan a ambos apuntadores . Todas las demás operaciones aritméticas están prohibidas. En particular, no se pueden multiplicar o dividir apuntadores; no se pueden sumar o restar apuntadores; no se les puede aplicar el desplazamiento a nivel de bits y los operadores de máscara; y no se puede sumar o restar el tipo float o el tipo double a los apuntadores .

5.2.7. Comparación de apuntadores .

Se pueden comparar dos apuntadores en una expresión relacional. Por ejemplo, dados dos apuntadores p y q, la siguiente instrucción es perfectamente válida:

```
if(p < q)
  printf("p apunta a menor memoria que q\n");
```

Generalmente, la comparación de apuntadores resulta útil cuando dos apuntadores apuntan a un objeto común, como puede ser un arreglo.

5.2.8. Apuntadores y arreglos

Existe una estrecha relación entre los apuntadores y los arreglos. Considérese el siguiente fragmento:

```
char cad[80] , *p1;

p1 = cad;
```

Aquí a p1 le ha sido asignada la dirección del primer elemento del arreglo cad. Para acceder al quinto elemento de cad se escribe

```
cad[4]  ó  *(p1+4)
```

Ambas instrucciones devuelven el quinto elemento. Recuérdese que los arreglos comienzan en el elemento 0. Para acceder al quinto elemento se debe usar 4 como índice de cad. También se puede sumar 4 al apuntador p1 para acceder al quinto elemento, porque p1 actualmente apunta al primer elemento de cad. (Recuérdese que un nombre de arreglo sin índice devuelve la dirección constante de comienzo del arreglo, que es el primer elemento).

El ejemplo anterior se puede generalizar. En esencia, C proporciona dos métodos de acceso a elementos de arreglos: la aritmética de apuntadores y la indexación de arreglo. Aunque la notación normal de indexación de arreglo a veces es más fácil de entender, la aritmética de apuntadores puede ser más rápida. Como la velocidad es muy considerada frecuentemente en programación, es común el uso de apuntadores para acceder a elementos de arreglos en los programas en C.

Los siguientes ejemplos ilustran lo anterior:

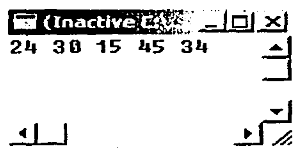
Programa 36

```
/* En este programa accedemos a los elementos de un arreglo mediante la indexación de arreglos
*/

# include <stdio.h>

void main(void)
{
   static int lista[] = {24,30,15,45,34};
   int ind;
   for (ind=0; ind<5; ind++)
        printf("%d ", lista[ind]);
}
```

Programa 37

```
/* En este programa accedemos a los elementos de un arreglo mediante la aritmética de
apuntadores */

# include <stdio.h>

void main(void)
{
   static int lista[] = {24,30,15,45,34};
   int ind;

   for (ind=0; ind<5; ind++)
        printf("%d ",*(lista+ind));
}
```

- Los dos programas imprimen la misma salida utilizando métodos diferentes para acceder a los elementos del arreglo.
- Su salida es la siguiente.



5.2.9 Arreglos de apuntadores

Los apuntadores pueden estructurarse en arreglos como cualquier otro tipo de datos. La declaración para un arreglo de apuntadores a enteros (int) de tamaño 10 es:

```
int *x[10];
```

Para acceder al asignar la dirección de una variable entera llamada var al tercer elemento del arreglo de apuntadores se escribe

```
x[2] = &var;
```

Y para encontrar el valor de var

```
*x [2];
```

Si se quiere pasar un arreglo de apuntadores a una función, se puede utilizar el mismo método que se utiliza para otros arreglos: llamar simplemente a la función con el nombre del arreglo sin índices. Por ejemplo, una función que reciba el arreglo x sería como ésta:

```
void mostrar arreglo(int *q[])

{
int t;
for (t=0; t<10; t++)
    printf("%d ', *q[t]);
}
```

Obs.- Esta función es igual que las que hemos visto anteriormente con la diferencia de que esta tiene un nombre diferente de main y recibe un argumento. Mas adelante veremos las funciones mas a detalle.

Recuérdese que q no es un apuntador a enteros, sino un apuntador a un arreglo de apuntadores a enteros. Por tanto, hay que declarar el parámetro q como un arreglo de apuntadores a enteros como se muestra en el código anterior.

Los arreglos de apuntadores frecuentemente se utilizan para mantener apuntadores a mensajes de error. Se puede crear una función que muestre un mensaje dado su número de código de error, tal como se muestra a continuación:

```
void error de sintaxis(int num)

static char *err[]= {
        "No se puede abrir el archivo\n",
        "Error de lectura\n",
        "Error de escritura\n",
        "Fallo del dispositivo\n"
    }

printf("%s", err[num]);
}
```

El arreglo err guarda los apuntadores a cada cadena de error. Esto funciona porque una constante de cadena que se usa en una expresión (una inicialización, en este caso) produce un apuntador a la cadena. Se llama a la función printf() con un apuntador a carácter que apunta al mensaje de error indexado por el número de error pasado a la función. Por ejemplo, si num pasa un 2, entonces se muestra el mensaje "Error de escritura".

Es interesante fijarse que el argumento de línea de órdenes argv es un arreglo de apuntadores a carácter.

5.2.10. Indirección múltiple

Se puede hacer que un apuntador apunte a otro apuntador que apunte a una variable de destino. Esta situación se denomina indirección múltiple o apuntadores a apuntadores. Los apuntadores a apuntadores pueden resultar confusos. La **Figura 5.2** ilustra el concepto de indirección múltiple. Como se puede ver, el valor de un apuntador normal es la dirección del objeto que contiene el valor deseado. En el caso de un apuntador a apuntador, el primer apuntador contiene la dirección del segundo apuntador, que apunta al objeto que contiene el valor deseado.

La indirección múltiple puede llevarse a la extensión que se desee, pero existen pocos casos en los que se necesite más de un apuntador a apuntador. De hecho, la indirección excesiva es difícil de seguir y propensa a errores conceptuales.

Una variable que es un apuntador a apuntador tiene que declararse como tal. Esto se hace colocando un * adicional delante del nombre de la variable. Por ejemplo, la siguiente declaración indica al compilador que Casa es un apuntador a apuntador de tipo float.

```
float **Casa;
```

Es importante entender que casa no es un apuntador a un número real, sino un apuntador a un apuntador a float. Para acceder al valor de destino indirectamente apuntado por un apuntador a apuntador hay que poner dos veces el operador asterisco, como en este ejemplo:

Programa 38

```
/* En este programa se realiza una indirección múltiple */

#include <stdio.h>

void main(void)
{
 int x, *p, **q;

 x = 10;
 p = &x;
 q = &p;
 printf("%d", **q); /* imprime el valor de x */
 }
```

- Aquí p se declara como un apuntador a entero y q como un apuntador a apuntador a entero. La llamada a printf() imprime 10 en la pantalla.

Figura 5.2

Apuntador Variable

dirección ---- > valor

Indirección simple

Apuntador Apuntador Variable

dirección -----> dirección ------> valor

Indirección múltiple

5.2.11. Inicialización de apuntadores

Después de declarar un apuntador local no estático, pero antes de asignarle un valor, contiene un valor desconocido. (Los apuntadores locales estáticos y globales se inicializan automáticamente a nulo). Si se intenta utilizar el apuntador antes de darle valor, probablemente se estrellará el programa -y posiblemente también el sistema operativo de la computadora-, ¡un error bastante desagradable!

Existe un importante convenio que siguen la mayoría de los programadores de C cuando trabajan con apuntadores : Un apuntador que no apunte a ninguna posición válida de la memoria ha de tener asignado el valor nulo (que es un cero). Se usa el nulo porque C garantiza que no existe ningún objeto en la dirección nula. Así, cualquier apuntador que sea nulo indica que no apunta a nada y no debe ser usado.

Una forma de dar a un apuntador el valor nulo es asignándole cero. Por ejemplo, lo siguiente inicializa p a nulo:

```
char *p = 0;
```

Adicionalmente, muchos de los archivos de cabecera de C, como <stdio.h>, definen la macro NULL, una constante de apuntador nulo. Por eso, a menudo se verá la inicialización de un apuntador a nulo con una instrucción como ésta:

```
p = NULL;
```

Sin embargo, el hecho de que un apuntador contenga un valor nulo no significa necesariamente que sea "seguro". El uso del valor nulo es simplemente un convenio que siguen los programadores. No es una regla impuesta por el lenguaje C. Por ejemplo, la secuencia que sigue, aunque incorrecta, no provocará ningún error de compilación:

```
int *p = 0;
*p = 10; /* ¡incorrecto! */
```

En este caso, la asignación a través de p provoca una asignación en 0, lo que normalmente hará que se estrelle el programa.

Como se supone que no se va a utilizar un apuntador nulo, se puede utilizar el apuntador nulo para hacer muchas de las rutinas de apuntadores más fáciles de codificar y más eficientes.

Por ejemplo, se puede utilizar un apuntador nulo para marcar el final de un arreglo de apuntadores. Una rutina que acceda al arreglo sabe que ha llegado al final cuando encuentra el valor nulo.

Es una práctica común en los programas en C se inicializan los apuntadores char * de forma que apunten a constantes de cadena. Para entender cómo funciona ésto, considérese la siguiente instrucción:

```
char *p = "hola amigo";
```

Como se puede ver, p es un apuntador, no un arreglo. Esto hace que surja una pregunta: ¿Dónde se encuentra almacenada la constante de cadena "hola amigo"? Como p no es un arreglo, no puede estar guardada en p. Pero, obviamente, la cadena tiene que estar guardada en algún sitio. La respuesta a esa pregunta se encuentra en la forma que tiene el compilador de manejar las constantes de cadena. El compilador de C crea lo que se llama una *tabla de cadenas,* en la que se guardan las constantes de cadena utilizadas por el programa. Por tanto, la anterior instrucción de declaración pone en el apuntador p la dirección que tiene "hola amigo" en la tabla de cadenas. Hasta el final del programa, p puede utilizarse como cualquier otra cadena. Por ejemplo, el programa que sigue es perfectamente válido:

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

char*p = "hola amigo";
int main(void)
{

 register int t;

/* imprimir la cadena hacia delante y hacia atrás */
printf(p);
for(t=strlen(p)-1; t>-1; t--)
    printf("&c", p[t]);

return 0;
}
```

5.2.12. Problemas con apuntadores

¡Nada puede dar más problemas que un apuntador descontrolado! Los apuntadores son un arma de doble filo. Proporcionan una potencia tremenda, pero cuando un apuntador se utiliza incorrectamente, o contiene un valor equivocado, el error puede ser de lo más difícil de encontrar.

Un error de apuntador es difícil de localizar porque el problema no es el apuntador en sí mismo. El problema surge cuando se accede a un objeto a través del apuntador. Sencillamente, cada vez que se utiliza un apuntador incorrecto, se lee o escribe en algún lugar desconocido de la memoria. Si se lee

con el puntero, lo que ocurre es que se obtiene basura, lo que probablemente hará que el programa no funcione bien. Si se escribe con él, puede que se esté escribiendo en otras partes de código o datos. En cualquier caso, el problema puede no evidenciarse hasta más tarde durante la ejecución del programa, lo que puede llevar a buscar el error en un lugar equivocado. Puede que no haya nada o casi nada que sugiera que la causa original del problema es el apuntador. Este tipo de errores hace que los programadores pierdan el sueño y el tiempo una y otra vez.

Como los errores de apuntadores se convierten en pesadillas, hay que esforzarse para que no se genere ninguno. Para ayudar a evitarlos, aquí se comentan algunos de los errores más comunes. El ejemplo clásico de un error de apuntador es el apuntador no inicializado. Considérese este programa:

```
main (void)
 {
int x, *p;
x = 10;
*p = x;    /* error: p no se ha inicializado */
 return 0;
   }
```

Este programa asigna el valor 10 a alguna posición de memoria desconocida. ¿Por qué? Como el apuntador p nunca ha recibido un valor, contiene un valor desconocido cuando se lleva a cabo la asignación *p=x. Esto hace que se escriba el valor x en alguna posición indeterminada de la memoria. Este tipo de problema a menudo pasa inadvertido cuando el programa es pequeño, debido a que las posibilidades están a favor de que p contenga una dirección "segura"; una dirección que no es del código, del área de datos o del sistema operativo. Sin embargo, a medida que el programa crece, aumenta la probabilidad de que p apunte a algo vital. Eventualmente, el programa se para. Para este sencillo ejemplo, la mayoría de los compiladores mostrarán un mensaje de advertencia que indica que se está intentando utilizar un apuntador que no ha sido inicializado, pero este mismo tipo de error se puede producir de otras formas más rebuscadas que el compilador no puede detectar.

Un segundo error común se produce por un simple desconocimiento del uso de los apuntadores . Considérese lo siguiente:

```
#include <stdio.h>
int main(void)
{
int x,*p
x = 10 ;
*p = x;
printf("%d",*p);
return 0;
}
```

La llamada a printf() no imprime el valor de x en la pantalla, que es 10. Imprime algún valor desconocido porque la asignación

```
*p = x;
```

está mal. La instrucción asigna el valor 10 al apuntador p. Sin embargo, se supone que p contiene una dirección, no un valor. Para corregir el programa hay que escribir

```
p = &x;
```

Como con el error anterior, la mayoría de los compiladores mostrarán al menos un mensaje de advertencia al intentar asignar x a p. Pero igual que antes, este error se puede manifestar de una forma sutil que el compilador no pueda detectar.

Otro error que ocurre a veces se produce por una idea incorrecta sobre cómo se sitúan las variables en memoria. En general, no se puede saber dónde se colocarán los datos en la memoria, si se situarán de nuevo en el mismo lugar o incluso si se tratarán del mismo modo por compiladores diferentes. Por esta razón, la comparación entre apuntadores que no apuntan al mismo objeto produce resultados inesperados. Por ejemplo,

```
char c[80], y[80];
char *p1, *p2;
p1 = c;
p2 = y;
if (p1 < p2)
```

es generalmente un concepto erróneo. (En muchas situaciones inusuales se podría utilizar algo como esto para determinar la posición relativa de las variables. Pero se trata de algo más bien raro).

Cuando se asume que dos arreglos adyacentes pueden ser indexados como uno simplemente incrementando el apuntador a través de los límites del arreglo, se produce un error similar. Por ejemplo,

```
int primero[10], segundo[10];
int *p, t;
p = primero;
for(t=0;t<20;++t)
    *p++=t ;
```

Ésta no es una buena forma de inicializar los arreglos primero y segundo con los números del 0 al 19. Aun suponiendo que puede funcionar en algún compilador bajo ciertas circunstancias, asume que ambos arreglos se colocan uno tras otro en memoria con primero el primero. Éste no es siempre el caso.

El siguiente programa ilustra un tipo de error muy peligroso. Véase si se puede encontrar.

```
/* Este programa tiene un error. */

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

    int main(void)
    {
    char *p1;
    char c[80],
    p1 = c;
    do
    {
     gets(c);     /* leer una cadena */

  /*imprimir el equivalente decimal de cada carácter */

     while(*p1)
         printf(" %d", *p1++);
    }
     while(strcmp(c, "hecho"));
     return 0;
    }
```

Este programa usa p1 para imprimir los valores ASCII asociados con los caracteres contenidos en c. El problema es que a p1 sólo se le ha asignado la dirección de c una vez. La primera vez que se entra en el ciclo, p1 apunta al primer carácter de C. Sin embargo, la segunda vez que se entra, continúa donde se quedó ya que no se ha reasignado al comienzo de c. ¡Ese carácter siguiente puede ser parte de la segunda cadena, de otra variable o de una parte de programa! La forma correcta de escribir el programa es:

```
/*Ahora, este programa es correcto. */

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

int main(void)

char *p1;
char c[80];

do
{
  p1 = c;          /* colocar p1 al principio de c */
  gets(c);        /* leer una cadena */

 /*imprimir el equivalente decimal de cada carácter */

   while(*p1)
        printf(" %d", *p1++);
 }
   while(strcmp(c, "hecho"));

 return 0;
 }
```

Aquí, cada vez que itera el ciclo, p1 se coloca al principio de la cadena. En general, hay que acordarse de reinicializar un apuntador si es que se va a reutilizar.

El hecho de que el incorrecto manejo de los apuntadores pueda producir errores engañosos no es motivo para evitar su uso. Simplemente hay que tener cuidado y asegurarse de que se sabe a dónde apunta cada apuntador antes de usarlo.

5.3. Funciones

Las funciones son los bloques constructores de C y el lugar donde se produce toda la actividad del programa. Esta sección examina sus características, incluyendo los argumentos, los valores devueltos, los prototipos y la recursión.

5.3.1. Forma general de una función

La forma general de una función es:

```
tipo_ dev   nombre_de_ la_función (lista de parámetros)
{
   cuerpo de la función
}
```

El tipo_dev especifica el tipo de dato que devuelve la función. Una función puede devolver cualquier tipo de dato excepto un arreglo. La lista de parámetros es una lista de nombres de variable separados por comas con sus tipos asociados. Los parámetros reciben los valores de los argumentos cuando se llama a la función. Una función puede no tener parámetros, en cuyo caso la lista de parámetros está vacía. Una lista de parámetros vacía se puede especificar explícitamente como tal colocando la palabra clave void entre los paréntesis.

En las declaraciones de variables se pueden declarar múltiples variables del mismo tipo mediante una lista con los nombres de las variables separados por comas. En cambio, en las funciones todos los parámetros deben declararse individualmente, incluyendo para cada uno tanto el tipo como el nombre. Es decir, la lista de declaración de parámetros de un función tiene la siguiente forma general:

```
F (tipo varl, tipo var2, ..., tipo varN)
```

Por ejemplo, a continuación se muestran una declaración correcta de parámetros de función y otra incorrecta:

```
f(int i, int k, int j)     /* correcto */

f(int i, k, float j)       /* incorrecto: k debe tener su propio tipo */
```

5.3.2. El alcance de una función

Las reglas de alcance de un lenguaje son las reglas que controlan si un fragmento de código conoce o tiene acceso a otro fragmento de código o de datos.

Cada función es un bloque de código discreto. Por tanto, una función define un alcance de bloque. Esto significa que el código de una función es privado a esa función y no se puede acceder a él mediante una instrucción de otra función, a menos que se haga a través de una llamada a esa función. (No es posible, por ejemplo, utilizar un go to para saltar en medio de otra función). El código que comprende el cuerpo de una función está oculto al resto del programa y, a no ser que se usen datos o variables globales, no puede ser afectado por otras partes

del programa ni afectarlas. Dicho de otro modo, el código y los datos que están definidos dentro de una función no pueden interactuar con el código o los datos definidos dentro de otra función porque las dos funciones tienen alcances diferentes.

Las variables que están definidas dentro de una función son variables locales. Una variable local comienza a existir cuando se entra en la función y se destruye al salir de ella. Así, las variables locales no pueden conservar sus valores entre distintas llamadas a la función. La única excepción a esta regla se da cuando la variable se declara con el especificador de clase de almacenamiento static. Esto hace que el compilador trate a la variable como si fuese una variable global por lo que al almacenamiento se refiere, pero manteniendo su alcance limitado a la función.

Los parámetros formales de una función también tienen la función como alcance. Esto significa que los parámetros se conocen en toda la función. Un parámetro comienza a existir cuando se entra en la función y se destruye al salir de ella.

Todas las funciones tienen alcance de archivo. Es decir, no se puede definir una función dentro de otra función. Esto es por lo que C no es técnicamente un lenguaje estructurado en bloques.

5.3.3. Argumentos de funciones

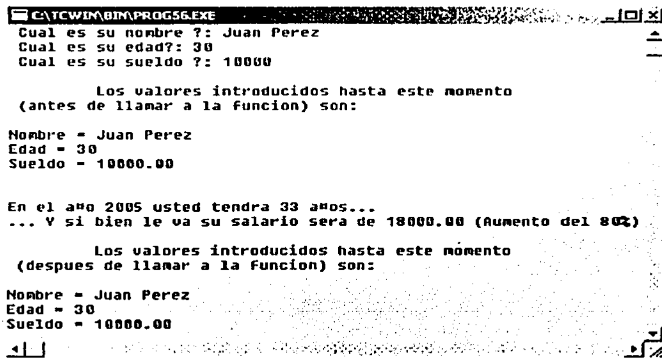
Si una función va a aceptar argumentos, debe declarar variables que reciban los valores de los argumentos. Como muestra la siguiente función, la declaración de parámetros va detrás del nombre de la función.

Programa 39

```
/* Este programa llama a una función la cual contiene 2 parámetros que el usuario introduce mediante el
teclado */

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

void Incrementa_Sueldo(int, float);     /* Prototipo de función */

void main(void)
{
char Nombre[35];
int Edad;
float Sueldo;

clrscr();
printf(" Cual es su nombre ?: ");
```

```
gets(Nombre);
printf(" Cual es su edad?: ");
scanf("%d", &Edad);
printf(" Cual es su sueldo ?: ");
scanf("%f", &Sueldo);
printf("\n\t Los valores introducidos hasta este momento \n (antes de llamar a la función) son: \n");
printf("\nNombre = %s", Nombre);
printf("\nEdad = %d", Edad);
printf("\nSueldo = %8.2f\n\n", Sueldo);
Incrementa_Sueldo(Edad, Sueldo);
printf("\n\t Los valores introducidos hasta este momento \n (después de llamar a la   función) son: \n");
printf("\nNombre = %s", Nombre);
printf("\nEdad = %d", Edad);
printf("\nSueldo = %8.2f\n\n", Sueldo);
getch();
}
void Incrementa_Sueldo(int Edad, float Sueldo)
{
Edad=Edad+3;
Sueldo=Sueldo*1.8;
printf("\nEn el año 2005 usted tendrá %d años...", Edad);
printf("\n... Y si bien le va su salario será de %8.2f (Aumento del 80%%)\n", Sueldo);
getch();
}
```

- Una salida de esta función es:



- En este programa los parámetros Edad y Sueldo utilizados por la función son proporcionados por el usuario. ¿por qué no cambian estos después de llamar a la función?

- Observa con atención las líneas que aparecen en el código de programa en negritas. La primera nos indica que usaremos la función incrementa_sueldo en el programa, la segunda manda llamar a la función y la tercera junto con su bloque de código es la función (donde se realizas todas las operaciones de esta).

Aun cuando los parámetros realizan la tarea especial de recibir el valor de los argumentos que se pasan a la función, se comportan como cualquier otra variable local. Por ejemplo, se pueden hacer asignaciones a los parámetros formales de una función o usarlos en cualquier expresión.

5.3.4. Llamada por valor y por referencia

En los lenguajes de computadora se contemplan dos formas de paso de argumentos a las funciones . El primer método se denomina llamada por valor. Este método copia el valor de un argumento en el parámetro formal de la función. En este caso, los cambios realizados sobre el parámetro no afectan al argumento.

La llamada por referencia es la segunda forma de paso de argumentos a subrutinas. Con este método se pasa la dirección del argumento. Dentro de la función se usa la dirección para acceder directamente al argumento usado en la llamada. Esto significa que los cambios sufridos por el parámetro se reflejan en el argumento.

Con unas pocas excepciones, C utiliza la llamada por valor para el paso de argumentos. Esto significa, en general, que el código de la función no puede alterar los argumentos usados en la llamada a la función. Considérese el siguiente programa:

Programa 40

```
/* Este programa crea una llamada por valor a una función */

# include <stdio.h>

int cuad(int x);

int main (void)
    {
```

```
int t=10;
printf("%d %d", cuad(t), t);
return 0;
        }
int cuad(int x)
        {
x = x*x;
return( x);
        }
```

- En este ejemplo se copia el valor del argumento de cuad(), 10, en el parámetro x. Cuando se realiza la asignación x = x*x, el único elemento que se modifica es la variable local x. La variable t, usada para llamar a cuad(), sigue teniendo el valor 10. De esta forma, la salida es 100 10.

Recuérdese que lo que se pasa a la función es una copia del valor del argumento. Lo que ocurra dentro de la función no tiene efecto sobre la variable utilizada en la llamada.

5.3.4.1. Creación de una llamada por referencia

Aunque el convenio de paso de parámetros de C es la llamada por valor, se puede crear una llamada por referencia pasando un apuntador al argumento, en lugar de pasar el propio argumento. Como lo que se pasa a la función es la dirección del argumento, el código de la función puede cambiar el valor del argumento externo a la función.

Los apuntadores se pasan a las funciones como cualquier otro argumento. Por supuesto, es necesario declarar los parámetros como de tipo apuntador. Por ejemplo, la función inter(), que intercambia el valor de sus dos argumentos enteros, muestra cómo:

```
void inter(int *x, int *y)

int temp;

temp = *x;     /* guardar el valor de la dirección x */
*x = *y;       /* poner y en x */
*y = temp;     /* poner x en y */
```

- La función inter() puede intercambiar los valores de las dos variables apuntadas por x e y debido a que lo que se pasan son sus direcciones (no sus valores). Así, dentro de la función se acceden a los contenidos de las variables utilizando las operaciones de apuntadores usuales y los valores quedan intercambiados.

- Recuérdese que inter() (o cualquier otra función que utilice parámetros de tipo apuntador) debe ser llamada con las direcciones de los argumentos.

El siguiente programa muestra la forma correcta de llamar a inter():

Programa 41

```
/* Este programa crea una llamada por referencia a una función */

#include <stdio.h>
void inter(int *x, int *y);
int main(void)
{
 int i, j;
 i = 10;
 j = 20;
 printf(" i y j antes del intercambio: %d  %d \n", i, j);
 inter(&i, &j);       /* se pasan las direcciones de i y j */
 printf(" i y j tras el intercambio: %d %d \n", i, j);
 return 0;
}

void inter(int *x, int *y)
{
 int temp;
 temp = *x;       /*guardar el valor de la dirección x */
 *x = *y;         /*poner y en x */
 *y = temp;       /*poner x en y */
}
```

- La salida de este programa es la siguiente:

- En el programa, a la variable i se le asigna el valor 10 y a j el valor 20. Después se llama a inter() con las direcciones de i y de j. (Se utiliza el operador unario & para obtener la dirección de las variables). De esta manera, las direcciones de i y de j, no sus valores, son lo que se pasan a la función inter().

5.3.5. Llamada a funciones con arreglos

Los arreglos se explicaron con detalle en el sección 4.1. Sin embargo, ahora trataremos el paso de arreglos como argumentos debido a que se trata de una excepción al paso de parámetros por valor estándar.

Cuando se usa un arreglo como argumento de una función, sólo se pasa a la función la dirección del primer elemento del arreglo. Ésta es una excepción al convenio de paso de parámetro por valor. En este caso, el código que hay dentro de la función opera sobre el contenido real del arreglo usado para llamar a la función, pudiendo alterarlo. Por ejemplo, considérese la función imprimir mayúsculas(), que imprime su argumento de tipo cadena en mayúsculas:

Programa 42

```
/* Este programa pasa un arreglo como argumento de una función el cual es modificado dentro de la
misma */

#include <stdio.h>
#include <ctype.h>

void imprimir_mayúsculas(char *cadena);

 int main(void)
 {

  char c[80];

  printf("Introduce una cadena: ");
  gets(c);
  imprimir_mayúsculas(c);
  printf("\nc está ahora en mayúsculas: %s", c);
  return 0;
 }

/* Imprimir una cadena en mayúsculas */

  void imprimir_mayúsculas(char *cadena)
  {
```

```
  register int t;
  for (t=0; cadena[t]; ++t)
    {
    cadena[t] = toupper(cadena[t]);
    putchar(cadena[t]);
    }
  }
```

- La salida del programa es la siguiente:

```
(Inactive C:\TCWIN\BIN\PROG36.EXE)
Introduce una cadena: programar en c es facil
PROGRAMAR EN C ES FACIL
c está ahora en mayúsculas: PROGRAMAR EN C ES FACIL
```

- Después de la llamada a imprimir_mayúsculas(), el contenido del arreglo c de main() ha cambiado a mayúsculas.

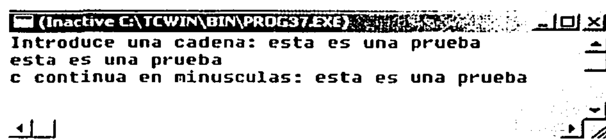
Si esto no es lo que se quiere que ocurra, se podría escribir el programa así:

Programa 43

```
/*Este programa pasa un arreglo como argumento de una función el cual no es modificado dentro de
la misma */

#include <stdio.h>
#include <ctype.h>

void imprimir_mayúsculas(char *cadena);

int main(void)
{

 char c[80];

 printf("Introduce una cadena: ");
 gets(c);
 imprimir_mayúsculas(c);
 printf("\nc continua en minúsculas: %s", c);
 return 0;
}
```

```
/* Imprimir una cadena en mayúsculas */

void imprimir_mayúsculas(char *cadena)
{
register int t;
for (t=0; cadena[t]; ++t)
   putchar(cadena[t]);
}
```

- Después de la llamada a imprimir_mayúsculas():



- El contenido del arreglo c permanece inalterado ya que sus valores no se modifican dentro de imprimir_mayúsculas(). ¿Podrías explicar esto?

5.3.6. La instrucción return

La instrucción tiene dos usos importantes. Primero, fuerza una salida inmediata de la función en que se encuentra, o sea, hace que se devuelva el control a la función que llama y en segundo lugar se puede utilizar para devolver un valor. Las secciones que siguen examinan las formas de aplicar esta instrucción.

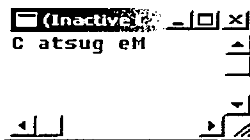
5.3.6.1. Vuelta de una función

Hay dos formas en las que una función puede terminar su ejecución y volver al sitio en que se llamó. La primera ocurre cuando se ha ejecutado la última instrucción de la función y, conceptualmente, se encuentra la llave de cierre } del final de la función. (La llave realmente no aparece en el código objeto, por supuesto, pero se puede pensar como si así fuera.) Por ejemplo, la función imp_inversa() de este programa sencillamente imprime la cadena "Me gusta C" invertida en la pantalla y luego vuelve.

Programa 44

```
/* Este programa nos ilustra como regresar de una función */

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

void imp_inversa(char *s);

int main(void)
{
 imp_inversa("Me gusta C");
 return 0;
}
void imp_inversa(char *s)
{
 register int t;

 for(t=strlen(s)-1; t>=0; t--)
 putchar(s[t]);
}
```

- La salida del programa es:



- Una vez que ha impreso la cadena, la función imp_inversa() no tiene nada más que hacer y devuelve el control al lugar que la llamó.

Actualmente no hay muchas funciones que utilicen esta forma implícita de terminación. La mayoría de las funciones emplean la instrucción return para terminar la ejecución, bien porque se tiene que devolver un valor o bien para simplificar y hacer el código más eficiente. Recuérdese que una función puede tener varias instrucciones return.

5.3.6.2. Valores devueltos

Todas las funciones excepto aquellas de tipo void, devuelven un valor. Este valor se especifica explícitamente en la instrucción return. En C89, si una función que no sea de tipo void ejecuta una instrucción return sin especificar un valor, entonces se devuelve un valor sin sentido. Como poco, se puede decir que es ¡una mala práctica! En C99 (y en C++), una función que no sea de tipo void debe devolver un valor por medio de una instrucción return. Es decir, en C99, si se especifica una función como devolviendo un valor, cualquier instrucción return que contenga la función debe tener un valor asociado.

Sin embargo, si la ejecución de una función que no sea void llega al final de la función (esto es, encuentra la llave de cierre del final de la función), se devuelve basura como valor de vuelta. Aunque esta situación no constituye un error de sintaxis, sigue siendo una mala práctica y se debe evitar.

Mientras que una función no se declare como void, puede ser usada como operando en cualquier expresión. Por tanto, cada una de las siguientes expresiones es válida:

- ✓ z= potencia(y);
- ✓ if (max(x,y) > 100)
  printf("mayor");
- ✓ for (c=getchar(); isdigit(c);) .. ;

Como regla general, una función no puede ser el destino de una asignación. Una instrucción como:

```
inter(x,y) = 100;   /* instrucción incorrecta */
```

es incorrecta. El compilador de C marcará esta expresión como errónea y no compilará un programa que la contenga.

Cuando se escriben programas, las funciones serán de tres tipos.

- ✓ El primer tipo es simplemente computacional. Son funciones diseñadas específicamente para realizar operaciones con sus argumentos y devolver un valor basado en esos cálculos. Una función computacional es una función "pura". Ejemplos son las funciones de la biblioteca estándar sqrt() y sin(), que calculan la raíz cuadrada y el seno de sus argumentos.

- ✓ El segundo tipo de funciones manipula la información y devuelve un valor que indica simplemente el éxito o el fallo de esa manipulación. Un ejemplo es la función de biblioteca fclose(), que se usa para cerrar un archivo. Si la operación de cierre tiene éxito, la función devuelve 0; si se produce algún error, la función devuelve EOF.

✓ El último tipo de funciones no tienen un valor de vuelta explícito. En esencia, la función es estrictamente de tipo procedimiento y no genera un valor. Un ejemplo es exit(), que termina la ejecución de un programa. Todas las funciones que no devuelvan valores deben ser declaradas de tipo void. Al declarar una función de tipo void se evita que se uso en una expresión, previniendo así el mal uso accidental.

A veces, las funciones que no generan un resultado de interés devuelven algo de todas formas. Por ejemplo, printf() devuelve el número de caracteres escritos. Aun así, es muy raro encontrar un programa que lo utilice. En otras palabras, aunque todas las funciones, excepto las declaradas como de tipo void, devuelven valores, no se tiene que usar necesariamente ese valor de vuelta para algo. Una pregunta muy común acerca de los valores devueltos por las funciones es: "Ya que se devuelve un valor, ¿no se tiene que asignar ese valor de vuelta a alguna variable?". La respuesta es no. Si no hay una asignación especificada, el valor devuelto simplemente se ignora. Considérese el siguiente programa, que utiliza la función mul().

Programa 45

```
/* Este programa nos muestra como regresar el valor de una función */

#include <stdio.h>

int mul(int a, int b);

int main(void)
{
        int x, y, z;
        x = 10;
        y = 20;
        z = mul(x, y);              /* 1 */
        printf("%d", mul(x,y));     /* 2 */
        mul(x, y);                  /* 3 */
        return 0;
}

mul(int a, int b)
{
return a*b;
}
```

➢ En la línea 1, el valor de vuelta de mul() es asignado a z. En la línea 2, el valor de vuelta no se asigna, sino que se usa en la función printf(). Finalmente, en la línea 3, el valor de vuelta se pierde porque no se asigna a otra variable ni se usa como parte de una expresión.

- Si se imprime algún valor ¿cuál es?

5.3.6.3. Devolución de apuntadores

Aunque las funciones que devuelven apuntadores se manejan de la misma forma que cualquier otro tipo de función, hay que comentar unos cuantos conceptos importantes. Los apuntadores a variables no son ni enteros ni enteros sin signo. Son variables que guardan direcciones de otras variables. La razón de esta distinción está en el hecho de que la aritmética de los apuntadores depende del tipo base. Por ejemplo, si se incrementa un apuntador a un entero, éste contiene un valor que es cuatro unidades mayor que el valor anterior (asumiendo enteros de 4 bytes). En general, cada vez que se incrementa (o decrementa) un apuntador, éste apunta al siguiente (o anterior) elemento de datos de su tipo.

Como cada dato puede ser de longitud diferente, el compilador debe saber a qué tipo de dato está apuntando el apuntador. Por esta razón, una función que devuelva un apuntador ha de declarar explícitamente el tipo de apuntador que devuelve. Por ejemplo, ¡no se debe utilizar un tipo de vuelta de int * para devolver un apuntador char *! En algunos casos, una función necesitará devolver un apuntador genérico. En ese caso, el tipo de vuelta de la función debe especificarse como void *.

Para devolver un apuntador, una función debe ser declarada como teniendo un tipo apuntador de vuelta. Por ejemplo, la función encuentra devuelve un apuntador a la primera ocurrencia del carácter c en la cadena s: Si no se encuentra, devuelve un apuntador al terminado nulo.

Programa 46

```
/* Este programa nos enseña como devolver un apuntador a una función */

#include <stdio.h>

char *encuentra(char c, char *s);
int main(void)
{

char s[80], *p, c;
gets(s);
c = getchar();
p = encuentra(c, s);
if(*p)      /* se ha encontrado */
   printf("%s" , p);
else
```

```
printf("No se ha encontrado.");
return 0;
}
/*Devuelve un apuntador a la primera ocurrencia de c en s. */

char *encuentra(char c, char *s)
{
while (c!=*s && *s)
s++;
return(s);
}
```

- Este programa lee una cadena y después un carácter. Luego busca una ocurrencia del carácter en la cadena. Si el carácter está en la cadena, p apuntará a ese carácter y el programa imprimirá la cadena desde el punto de coincidencia. Cuando no se encuentra el carácter, p apuntará al terminador nulo, haciendo que *p sea falso. En este caso, el programa imprime No se ha encontrado.

5.4. Funciones de tipo void

Uno de los usos de void es el de declarar explícitamente funciones que no devuelven valores. Esto previene su uso en cualquier expresión y ayuda a descubrir un mal uso accidental . Algunas versiones antiguas de C no definen la palabra clave void. Por eso, en programas en C antiguos, las funciones que no devolvían valores simplemente eran implícitamente de tipo int, incluso aunque no se devolviera valor.

5.5. Lo que devuelve main()

La función main() devuelve un entero al proceso de llamada, que generalmente es el sistema operativo. Que main() devuelva un valor es lo mismo que llamar a exit() con ese mismo valor. Si main() no devuelve un valor explícitamente, el valor que se pasa al proceso de llamada queda técnicamente indefinido. En la práctica, la mayoría de los compiladores de C devuelven automáticamente 0, aunque no se debe confiar en ello cuando la portabilidad sea un aspecto importante.

5.6. Recursión

En C, las funciones pueden llamarse a sí mismas. En ese caso se dice que la función es recursiva. La recursión es el proceso de definir algo en términos de sí mismo y a veces se llama definición circular.

Un sencillo ejemplo de función recursiva es factr(), que calcula el factorial de un entero. El factorial de un número n es el producto de todos los números enteros entre 1 y n. Por ejemplo, el factorial de 3 es 1 X 2 X 3, o sea, 6. A continuación se muestra factr() y su equivalente iterativa:

```
/* recursiva */

int factr(int n);
{
 int resp;
 if (n= =1)
    return(1);
 resp = factr(n-1)*n;   /* llamada recursiva */
 return(resp);
}

/* no recursiva*/

int fact(int n)
{
int t, resp;
resp = 1;
for (t=1; t<=n; t++)
   resp=resp*(t);
return(resp);
}
```

- La versión no recursiva, fact(), debe estar clara. Utiliza un ciclo que empieza en 1 termina en n y que progresivamente multiplica cada número por el producto acumulado.

- La operación de la recursiva, factr(), es un poco más compleja. Cuando se llama a factr() con un argumento de 1, la función devuelve 1. En otro caso, devuelve el producto de factr(n-1)*n. Para evaluar esta expresión se llama a factr() con n-1. Esto ocurre hasta que n es igual a 1 y las llamadas a la función empiezan a volver.

- Al calcular el factorial de 2, la primera llamada a factr() produce una segunda llamada recursiva con 1 como argumento. Esta llamada devuelve 1, que se multiplica entonces por 2 (el valor original de n). La respuesta, entonces, es 2. Se puede probar y descubrir por uno mismo cómo funciona el factorial de 3. (Se puede encontrar interesante incluir instrucciones printf() en factr() para ver el nivel de cada llamada y cuáles son las respuestas intermedias).

Una llamada recursiva no produce una nueva copia de la función. Sólo los valores con los que se trabaja son nuevos. Al volver de una llamada recursiva se podría decir que las funciones recursivas tienen un efecto "telescópico" que las lleva fuera y dentro de sí mismas.

Aunque la recursión parece ofrecer la posibilidad de una mayor eficiencia, rara vez es ese el caso. Normalmente, las rutinas recursivas no minimizan significativamente el tamaño del código ni ahorran espacio de memoria. Además, la versiones recursivas de la mayoría de las rutinas se ejecutan un poco más despacio que sus versiones iterativas equivalentes, debido a la sobrecarga de las repetidas llamadas a la función.

La principal ventaja de las funciones recursivas es que se pueden usar para crear versiones de algoritmos más claras y más sencillas. Por último, alguna gente perece pensar más fácilmente de forma recursiva que de forma iterativa.

Cuando se escriben funciones recursivas, debe haber alguna instrucción condicional, como un if, en algún sitio que fuerce a la función a volver sin que se ejecute otra llamada recursiva. Si no se hace así, la función nunca devolverá el control una vez que se le ha llamado. Escribir funciones recursivas sin una instrucción condicional es un error muy común. Se puede utilizar libremente printf() durante el desarrollo del programa para poder ver qué es lo que pasa y poder interrumpir la ejecución si se detecta un error.

5.7. Prototipos de funciones

En los programas modernos escritos en C de forma apropiada, todas las funciones deben estar declaradas antes de ser utilizadas. Ésto normalmente se lleva a cabo mediante los prototipos de funciones. Los prototipos de funciones no eran parte del lenguaje C original, sino que fueron añadidos por el estándar C89. Aunque los prototipos no son técnicamente obligatorios, su uso se recomienda insistentemente. Los prototipos permiten que el compilador lleve a cabo una fuerte comprobación de tipos, de forma parecida a como lo hacen lenguajes como Pascal. Cuando se usan prototipos, el compilador puede encontrar a informar sobre conversiones de tipo ilegales entre el tipo de los argumentos usados en la llamada a la función y las definiciones de tipos de sus parámetros. El compilador también detectará diferencias entre el número de argumentos usados en la llamada a la función y el número de parámetros de la misma.

La forma general de un prototipo de función es

```
tipo nombre_de_la_función (tipo p1, tipo p2,..., tipo pN);
```

donde los pi son los parámetros.

El uso de los nombres de los parámetros es opcional. Sin embargo, permiten que el compilador identifique cualquier discordancia de tipo por su nombre, cuando se de un error, por lo que es una buena idea incluirlos.

Ejemplo:

```
/*Este programa usa un prototipo de función para forzar una fuerte comprobación de tipos. */

void cuad(int *i);      /* prototipo */

int main(void)
{
int x;
x = 10,
cuad(x);      /* discordancia de tipo */
return 0;
}
void cuad(int *i)
{
*i=*i**i;
}
```

- El programa ilustra el valor de los prototipos de funciones. Produce un mensaje de error debido a que contiene un intento de llamada a cuad() con un argumento entero en lugar del apuntador a entero que se requiere.

Una definición de función también sirve como prototipo de la misma, siempre que la definición aparezca antes de que se utilice por primera vez la función en el programa. Por ejemplo, el que sigue es un programa válido.

```
#include <stdio.h>

/* Esta definición también sirve como prototipo dentro de este programa. */

void f(int a, int b)
{
   printf("%d ", a % b);
}
int main(void)
{
   f(10,3);
   return 0 ;
}
```

En este ejemplo, como f() se ha definido antes de usarla en main(), no se requiere un prototipo aparte. Aunque en los programas pequeños la definición de

una función puede servir fácilmente como su prototipo, rara vez se puede hacer en los programas largos; especialmente cuando se encuentran distribuidos en varios archivos. Los programas de esta tesis incorporan prototipos aparte para las funciones porque en la práctica ésta es la forma en la que se escribe normalmente el código C.

La única función que no requiere prototipo es main(), debido a que es la primera función que llama cuando comienza el programa. En C, cuando una función no tiene parámetros, su prototipo usa void entre los paréntesis.

Por ejemplo, a continuación se muestra el prototipo de una función f() tal como aparecería en un programa en C.

```
float f(void);
```

Esto indica al compilador que la función no tiene parámetros y que cualquier llamada a la función que utilice argumentos será errónea.

Los prototipos de funciones permiten localizar errores antes de que se produzcan. Además, ayudan a verificar que el programa funcione correctamente, al no permitir que se llame a funciones con argumentos discordantes.

Una última cosa: Como las versiones iniciales de C no admiten la sintaxis completa de prototipos, éstos son totalmente opcionales en C. Esto es necesario para admitir código en C previo a los prototipos. Si se va a transportar código C a C++ es necesario completar los prototipos de las funciones para que el código se pueda compilar con éxito. Recuérdese que, aunque en C los prototipos son opcionales, en C++ son obligatorios. Esto significa que todas las funciones de un programa en C++ deben estar completamente prototipadas.

Por esto, la mayoría de los programadores de C incluyen los prototipos completos en sus programas.

5.7.1. Prototipos de funciones de la biblioteca estándar

Cualquier función de la biblioteca estándar que se utilice en un programa debe tener un prototipo. Para ello se debe incluir la cabecera (archivo de cabecera) apropiada para cada función de biblioteca. El compilador de C proporciona todas las cabeceras necesarias. En C, las cabeceras de biblioteca son (normalmente) archivos que utilizan la extensión .h.

Un archivo de cabecera contiene dos elementos principales: las definiciones utilizadas por las funciones de la biblioteca y los prototipos de las funciones de la biblioteca. Por ejemplo, <stdio.h> se incluye en la mayoría de los programas de esta tesis porque contiene el prototipo de printf().

5.8. Declaración de listas de parámetros de longitud variable

Se puede especificar una función con un número variable de parámetros. El ejemplo más común es printf(). Para indicar al compilador que se va a pasar a una función un número desconocido de argumentos, se debe terminar la declaración de sus parámetros con puntos suspensivos. Por ejemplo, el siguiente prototipo especifica que func() tendrá al menos dos parámetros enteros y un número desconocido (incluso ninguno) de parámetros adicionales.

```
int func (int a, int b, ...);
```

Esta forma de declaración también se usa en las definiciones de las funciones. Cualquier función que usa un número variable de argumentos debe tener al menos un argumento concreto. Por ejemplo, lo siguiente es incorrecto:

```
int func(...);   /* ilegal */
```

5.9. La regla del entero por default

La versión original de C incluía una característica que a veces se refiere como la regla de "entero por default" (también como regla de "entero implícito"). Esta regla establece que en ausencia de un especificador de tipo explícito se asume el tipo entero. Esta regla se incluyó en el estándar C89, pero ha sido eliminada en C99. (Tampoco está contemplada en C++). Como la regla de entero por default está ya obsoleta, no la usaremos. Sin embargo, como todavía se usa en muchos programas existentes, se debe conocer bien.

El uso más común de la regla de entero por default se encuentra en el tipo devuelto por las funciones. Hace años, muchos programadores de C (probablemente la mayoría) sacaban partido de esta regla cuando creaban funciones que devolvían un resultado de tipo entero. Así, hace años, una función como:

```
int f(void)
{
/*...*/
return 0;
}
```

a menudo se escribía así:

```
f(void) /* tipo devuelto: entero por default */
{
/*...*/
return 0;
}
```

- En la primera versión se especifica explícitamente entero como tipo devuelto. En la segunda se asume por default.

La regla de entero por default no sólo se aplica a los valores devueltos por las funciones (aunque es el mayor de sus usos). Recuérdese que la regla de entero por default no es admitida ni por C99 ni por C++. Por esto, su uso en los programas de C89 no se recomienda. Es mejor especificar explícitamente cada tipo utilizado en el programa.

Capítulo 6
C con aplicación Actuarial

Los programas que a continuación se presentan tienen la finalidad de ilustrar un poco algunas de las cosas que se pueden realizar con un lenguaje de programación. En estos se incluyen programas de Álgebra, Geometría, Matemáticas Financieras, Calculo Integral, Análisis Númerico, Análisis Matemático, Calculo Actuarial, Probabilidad y Estadística.

6.1. Álgebra Superior

Programa 47

```
/* Este Programa Calcula las raíces de un polinomio de segundo grado. */

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

void main(void)
{

float a, b, c, x1, x2, A;

clrscr();
printf("\n  Este Programa Calcula las raíces de un polinomio de segundo grado.");
printf("\n\n  Dame el valor del coeficiente del termino cuadrático: ");
scanf("%f", &a);
printf("  Dame el valor del coeficiente del termino lineal ");
scanf("%f", &b);
printf("  Dame el valor del coeficiente del termino independiente ");
scanf("%f", &c);
A=b*b-4*a*c;
if(A>=0)
  {
  printf("\n  Las raíces son:");
  printf("\n\n  x1=%f\n",(-b+sqrt(A))/(2*a));
  printf("  x2=%f\n",(-b-sqrt(A))/(2*a));
  }
else
```

```
  {
  printf("\n  Las raices son:");
  printf("\n\n  x1=%f+i%f\n",(-b/(2*a)),(sqrt(-A))/(2*a));
  printf("  x2=%f+i%f\n",(b/(2*a)),(sqrt(-A))/(2*a));
  }
getch();
}
```

Programa 48

```
/* Obtener un determinante de una matriz de 3x3*/

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

int main(void)
{

 float det[3][3];
 float auxiliar, determinante ;
 int filas, cols,f,c;

 clrscr();
 printf("\n  Este programa obtiene el determinante de una matriz de 3x3\n");
 printf("\n  Captura de la matriz \n\n\a");
 for (f=0; f<3; f++)
 {
   for (c=0; c<3; c++)
   {
       printf("  Matriz(%2d,%2d)=", f+1, c+1);
       scanf("%f",&auxiliar);
       det[f][c]=auxiliar;
   }
 }
 determinante=det[1][1]*det[2][2]*det[3][3]+
              det[1][2]*det[2][3]*det[3][1]+
              det[1][3]*det[2][1]*det[3][2]+
              det[1][3]*det[2][2]*det[3][1]-
              det[1][2]*det[2][1]*det[3][3]-
              det[1][1]*det[2][3]*det[3][2];

printf("\n  El Determinante es: %f ",determinante);
return(0);

}
```

Programa 49

```
/* Este programa calcula c a la potencia n, con c, un numero real y n un entero */

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

void main (void)
{

int n,n1,i;
float c,r;

printf("\n  Calcula c^n con c un numero real y n un entero");
printf("\n\n  Que numero quieres elevar a una potencia (c): ");
scanf("%f",&c);
printf("  A que potencia lo quieres elevar (n): ");
scanf("%d",&n);
r=1;
if(c==0 && n<=0)
  printf("\n  0 a la 0 no esta definido");
else
{
 if (n==0)
    r=1;
 else
 {
   n1=n;
   if(n<0)
   n=-n;
   for (i=1;i<n+1;i++)
     {
     r=r*c;
     }
     if(n1<0)
        r=1/r;
 }
 printf("El resultado es %f \n", r);
}
getch();

}
```

Programa 50

```
/*Este programa calcula n!*/

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

void main(void)
{

int i,n,p;
double  r;

do
{

 clrscr();
 r=1;

 printf("\n  A que numero le quieres sacar el factorial \n");
 printf("  recuerda que debe ser un natural:  ");
 scanf("%d",&n);
 if(n<0)
 {
  printf("\n\n  No se le puede sacar el factorial a un numero negativo \n");
 }
 else
 {
  for (i=1;i<n+1;i++)
    {
     r=r*i;
    }
   printf("\n\n  El factorial de %d es: %.0lf \n",n,r);
 }
 printf("\n  Si quieres salir coloca un 1\n  de lo contrario presiona otro numero:  ");
 scanf("%d",&p);
}
while( p!=1 );
}
```

6.2. Álgebra Lineal

Programa 51

```
/* Suma de matrices */

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

# define FILAS_MAX 10
# define COLS_MAX 10

void main(void)
{
  float matrizA[FILAS_MAX][COLS_MAX];
  float matrizB[FILAS_MAX][COLS_MAX];
  float matriz_suma[FILAS_MAX][COLS_MAX];
  float auxiliar;
  int filas, cols,f,c;

  clrscr();
  printf("\n  Dame el numero de filas de las matrices : ");
  scanf("%d",&filas);
  printf("  Dame el numero de columnas de las matrices : ");
  scanf("%d",&cols);
  if ((filas>FILAS_MAX) || (filas<1) || (cols>COLS_MAX) || (cols<1))
  {
    printf("\n  Las dimensiones de las filas o columnas deben ser de 1 a 10");
    exit(0);
  }

  /* Entrada de datos */
  printf("\nCaptura de la matriz A\n\n\a");
  for (f=0; f<filas; f++)
  {
    for (c=0; c<cols; c++)
    {
        printf("  MatrizA(%2d,%2d)=", f+1, c+1);
        scanf("%f",&auxiliar);
        matrizA[f][c]=auxiliar;
    }
  }
```

```
printf("\n\nCaptura de la matriz B\n\n\a");
for (f=0; f<filas; f++)
{
  for (c=0; c<cols; c++)
  {
      printf("  MatrizB(%2d,%2d)=", f+1, c+1);
      scanf("%f",&auxiliar);
      matrizB[f][c]=auxiliar;
  }
}

for (f=0; f<filas; f++)
{
  for (c=0; c<cols; c++)
      matriz_suma[f][c]=matrizA[f][c]+matrizB[f][c];
}

printf("\nSuma de las matrices A y B\n\a");
for (f=0; f<filas; f++)
{
  for (c=0; c<cols; c++)
      printf("\n Suma(%2d,%2d)= %.2f", f+1, c+1, matriz_suma[f][c]);
}
}
```

Programa 52

```
/*Este programa càlcula el producto de 2 matrices de a lo mas 10x10*/

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

void main ()
{
int a[10][10], b[10][10], c[10][10],i,j,k,l,m,n,p,q;

printf("\n  Calcula el producto de 2 matrices de a lo mas 10x10\n");
do
{
  printf("\n  Introduce el numero de columnas de la primera matriz:  ");
  scanf("%d",&n);
  printf("  Introduce el numero de renglones de la primera matriz: ");
  scanf("%d",&m);
  printf("  Introduce el numero de columnas de la segunda matriz:  ");
  scanf("%d",&q);
  printf("  Introduce el numero de renglones de la segunda matriz  ");
  scanf("%d",&p);

  if (n!=p)
  {
   clrscr();
   printf ("\n  Recuerda que el numero de columnas de la primera matriz \n");
   printf ("  debe ser igual al numero de renglones de la segunda matriz \n");
   printf ("  Vuelve a introducir tus datos \n\n");
  }

  if ( m>10 ||m<=0 || p>10 || p<=0 || q>10 || q<=0 || p>10 ||p<=0 )
  {
  clrscr();
  printf("\n  Las dimensiones de las matrices no pueden ser negativas \n");
  printf("  Introduce de nuevo tus datos \n");
  }
}
while (n!=p || m>10 ||m<=0 || p>10 || p<=0 || q>10 || q<=0 || p>10 ||p<=0  );

printf("\n");
for (i=0; i<m; i++)
{
  for(j=0; j<n; j++)
  {
    printf("  Dame la entrada (%d)(%d) de la primera matriz:  ", i+1, j+1);
```

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

 printf("\n");
 for (j=0; j<n; j++)
 {
   for (l=0; l<q; l++)
   {
     printf("  Dame la entrada (%d)(%d) de la segunda matriz:  " ,j+1,l+1);
     scanf("%d", &b[j][l]);
   }
 }
 clrscr ();
 printf("\n  La matriz A es:\n\n");
 for (i=0; i<m; i++)
 {
   for (j=0; j<n; j++)
     printf("  %d  " ,a[i][j]);
     printf("\n");
 }
 printf("\n  La matriz B es:\n\n");
 for (j=0; j<n; j++)
 {
 for (l=0; l<q; l++)
    printf("  %d  ", b[j][l]);
    printf("\n");
 }
 for (i=0; i<m; i++)
 {
    for (l=0; l<q; l++)
    {
      c[i][l]=0;
      for(j=0; j<n; j++)
      c[i][l]=c[i][l]+a[i][j]*b[j][l];
    }
 }
 printf("\n  La matriz C es:\n\n");   /*Muestra la matriz C*/
 for (i=0; i<m; i++)
 {
    for (j=0; j<q; j++)
    printf("  %d  " ,c[i][j]);
    printf("\n");
 }
 getch();
 }
```

6.3. Geometría Analítica

Programa 53

```
/* Este programa calcula el producto punto de  2 vectores de un tamaño<=10 */

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

void main(void)
{

float a[10],b[10];
int s, I , c=0;

clrscr();
printf("\n  ¿De que tamaño son tus vectores A y B:  ");
scanf("%d", &s);
printf("\n");
for(i=0;i<s; i++)
  {
  printf("  Dame el %d numero de tu vector A:  ",i + 1);
  scanf("%f", &a[i]);
  }
printf("\n");
for(i=0;i<s; i++)
  {
  printf("  Dame el %d numero de tu vector B:  ",i + 1);
  scanf("%f", &b[i]);
  }
for(i=0;i<s;i++)
  c=c+a[i]*b[i];
printf("\n  El producto punto de A y B es:  %d",c);
getch();
}
```

Programa 54

```
/*Este programa calcula la norma de un vector de tamaño<=10*/

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

void raiz( float c);

void main (void)
{

float a[10];
int s,i,c=0;

clrscr();
printf("\n  Cuantos elementos tiene tu vector :  ");
scanf("%d",&s);
printf("\n");
for(i=0;i<s;i++)
 {
 printf("  Dame el %d numero de tu vector :  ",i+1);
 scanf("%f",&a[i]);
 }
for(i=0;i<s;i++)
 c=c+a[i]*a[i];
raiz(c);
}

void raiz(float c)
{

float x1,b0,c0,x0,y,z;
int d=4,k;

clrscr();
x1=c;
if(c<0)
  c=-c;
if(c==0)
  b0=0;
else
  {

  if(c<1)
     b0=.0001;
  else
```

```
    {
    k=1;
    while (k*k<=c)
        k=k+1;
    b0=k-1;
    }
  c0=c/b0;
  z=1.0000;
  y=0.0000;
  while(z>y)
   {
    c0=(b0+c0)/2;
    b0=c/c0;
    z=c0-b0;
    if (z<0)
        z=-z;
    y=pow(10,-d);
    }
  }
if(x1<0)
  printf("\n  La norma del vector es:  %10.5f i",b0);
else
  printf("\n  La norma del vector es:  %10.5f ",b0);    /*imprimo la raíz de un no
negativo(real)*/
getch();

}
```

6.4. Matemáticas Financieras

Programa 55

```
/* Interés Compuesto */

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

void main (void)
{

float monto, i, total, x ;
int  n, z ;

printf("\n  Este programa te calcula un monto a interés compuesto \n");
printf("  que  genera  un  capital  en un tiempo x a una tasa i \n\n");
printf("  Dame el Capital:  ");
scanf("%f",&monto);
printf("  Dame el periodo en meses (entero positivo):  ");
scanf("%d",&n);
printf("  Dame la tasa de interés :  ");
scanf("%f",&i);
x=(1+i/100);
total=1;
 for (z=0;z<n;z++)
     total=total*x;
total = monto * total;
printf("\n  Tu monto es: %f", total);

}
```

6.4. Matemáticas Financieras

Programa 55

```
/* Interés Compuesto */

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

void main (void)
{

float monto, i, total, x ;
int  n, z ;

printf("\n  Este programa te calcula un monto a interés compuesto \n");
printf("  que  genera  un  capital  en un tiempo x a una tasa i \n\n");
printf("  Dame el Capital:  ");
scanf("%f",&monto);
printf("  Dame el periodo en meses (entero positivo):  ");
scanf("%d",&n);
printf("  Dame la tasa de interés :  ");
scanf("%f",&i);
x=(1+i/100);
total=1;
  for (z=0;z<n;z++)
     total=total*x;
total = monto * total;
printf("\n  Tu monto es: %f", total);

}
```

Programa 56

```
/* Calcula el monto y el valor presente de una Anualidad con n pagos
   periódicos a una tasa i */

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

int main(void)
{
 float monto,anu,n,x,x1,i,x2,anu2;

 printf("\n  Calcula el monto y el valor presente de una Anualidad con \n");
 printf("  n pagos periódicos a una tasa i\n\n");
 printf("  Cuantos pagos vas a realizar:  ");
 scanf("%f",&n);
 printf("  A que tasa:  ");
 scanf("%f",&i);
 printf("  Cual es el monto de tus pagos:  ");
 scanf("%f",&monto);
 x=(1+i/100);
 x1=pow(x,n);
 x2=pow(x,-n);
 anu2=monto*(1-x2)/(i/100);
 anu=monto*(x1-1)/(i/100);
 printf("\n  El monto de la anualidad es          :  %f\n",anu );
 printf("  El Valor Presente de la anualidad es:  %f",anu2);
 return(0);
}
```

6.5. Calculo Integral

Programa 57

```
/*Este programa calcula las sumas inferiores de Riemman de la función x2 para un
intervalo [a, b] Positivo */

#include <stdio.h>

void main (void)
{

float a, b, d, s i;
int n, I;

clrscr();
printf ("\n ¿Cuantos rectángulos quieres formar? ");
scanf ("%d", &n);
printf ("\n Dame el limite izquierdo del intervalo: ");
scanf ("%f", &a);
printf ("\n Dame el limite derecho del intervalo: ");
scanf ("%f", &b);
if (n>=0 & a>=0 & b>=0 & (a<=b))
  {
  d= (b-a)/n;
  si=0;
  for (i=0; i<n; i++)
        si= (a + i * d)*(a + i * d)*d + si;    /*Calcula el valor de si */
  printf ("\n La suma inferior es: %f \n", si);    /*Muestra el resultado de la suma
inferior */
  }
else
  {
  printf("\n El numero de rectángulos debe ser mayor o igual a cero, \n");
  printf("el intervalo debe ser positivo y el limite izquierdo menor que el derecho
\n");
  }

getch ();

}
```

6.6. Análisis Numérico

Programa 58

```
/*Esta programa calcula la raiz cuadrada de x */

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

void main (void)

{

float x1,x,b0,c0,x0,y,z;
int d=4,k;

printf("\n  Este programa calcula la raiz cuadrada de un numero x por \n");
printf("  aproximaciones \n\n" );
printf("  A que numero le quieres sacar la raíz cuadrada:  ");
scanf("%f",&x);
x1=x;
if(x<0)
  x=-x;
if(x==0)
  b0=0;
else
  {

  if(x<1)
    b0=.0001;
  else
    {
    k=1;
    while (k*k<=x)
        k=k+1;
    b0=k-1;
    }
  c0=x/b0;
  z=1.0000;
  y=0.0000;
  while(z>y)
   {
    c0=(b0+c0)/2;
```

```
    b0=x/c0;
    z=c0-b0;
    if (z<0)
        z=-z;
    y=pow(10,-d);
    }
  }
if(x1<0)
  printf("\n  La raíz cuadrada de %f es: %10.5f i",x1,b0);
else
  printf("\n  La raíz cuadrada de %f es: %10.5f ",x,b0);
}
```

Programa 59

```
/* El método de Newton dice:
     ri+1 = (n / ri + rj) / 2
   La raíz cuadra calculada será valida, cuando se cumpla que
     abs(ri - ri+1) <= epsilon   */

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

void main(void)
{
  double numero;
  double aprox;
  double antaprox;
  double epsilon;

  printf("\n  Este programa te calcula la raíz cuadrada con el\n");
  printf("  Método de Newton\n\n");
  printf("  Dame el numero al que le quieres sacar la raíz cuadrada: ");
  scanf("%lf", &numero);
  printf("  Dame la raíz cuadrada aproximada: ");
  scanf("%lf", &aprox);
  printf("  Dame el coeficiente de error: ");
  scanf("%lf", &epsilon);
  do
  {
    antaprox=aprox;
    aprox=(numero/antaprox+antaprox)/2;
  }
  while(fabs(aprox-antaprox) >= epsilon);
  printf("\n\n  La raíz cuadrada de %.2f es %.2f", numero, aprox);
}
```

6.7. Análisis Matemático

Programa 60

```
/* Convergencia de la Serie Up Dawn */

#include<stdio.h>

void main (void)
{

int s,i,j;
printf("\n  Calcula la convergencia de la serie Up Dawn\n\n");
printf("  Dame el valor inicial:  ");
scanf("%d",&s);
i=0;

while((s!=0))
  {
  i++;
  if((s/2)==0)
    s=s/2;
  else
    s=3*s+1;
  }
printf("\n  El numero apartir del cual la serie converge a uno es:=%d",i);
getch();
}
```

6.8. Calculo Actuarial

Programa 61

```
/* Calcula conmutados */

# include <stdio.h>

int main(void)
{
int edad, i,lineas=1;
float lx[102],dx[102],px[102],radix=100000000;
float qx[102] [2]={

0 ,     0.006525000,
1 ,     0.000693000,
2 ,     0.000693000,
3 ,     0.000693000,
4 ,     0.000693000,
5 ,     0.000693000,
6 ,     0.000693000,
7 ,     0.000693000,
8 ,     0.000693000,
9 ,     0.000693000,
10,     0.000693000,
11,     0.000693000,
12,     0.000720000,
13,     0.000776000,
14,     0.000836000,
15,     0.000900000,
16,     0.000970000,
17,     0.001044000,
18,     0.001125000,
19,     0.001212000,
20,     0.001304000,
21,     0.001405000,
22,     0.001513000,
23,     0.001630000,
24,     0.001756000,
25,     0.001891000,
26,     0.002036000,
27,     0.002193000,
28,     0.002362000,
29,     0.002543000,
30,     0.002739000,
31,     0.002950000,
32,     0.003177000,
33,     0.003422000,
34,     0.003685000,
```

```
35,    0.003969000,
36,    0.004274000,
37,    0.004603000,
38,    0.004956000,
39,    0.005337000,
40,    0.005747000,
41,    0.006189000,
42,    0.006663000,
43,    0.007175000,
44,    0.007725000,
45,    0.008318000,
46,    0.008956000,
47,    0.009642000,
48,    0.010380000,
49,    0.011174000,
50,    0.012028000,
51,    0.012947000,
52,    0.013936000,
53,    0.014999000,
54,    0.016142000,
55,    0.017370000,
56,    0.018692000,
57,    0.020113000,
58,    0.021638000,
59,    0.023123000,
60,    0.024885000,
61,    0.026934000,
62,    0.028967000,
63,    0.031148000,
64,    0.033491000,
65,    0.036003000,
66,    0.038699000,
67,    0.041588000,
68,    0.044687000,
69,    0.048007000,
70,    0.051563000,
71,    0.055372000,
72,    0.059447000,
73,    0.063806000,
74,    0.068467000,
75,    0.073447000,
76,    0.078765000,
77,    0.084441000,
78,    0.090494000,
79,    0.096944000,
80,    0.103814000,
81,    0.111122000,
82,    0.118890000,
83,    0.127140000,
84,    0.135893000,
85,    0.145168000,
```

```
86,     0.154987000,
87,     0.165369000,
88,     0.176329000,
89,     0.187884000,
90,     0.200051000,
91,     0.212841000,
92,     0.226262000,
93,     0.240323000,
94,     0.255027000,
95,     0.270372000,
96,     0.286354000,
97,     0.302964000,
98,     0.320189000,
99,     0.664476000,
100,    1.000000000 } ;

printf("\n  Dadas las qx y un radix (de 10,000,000)\n");
printf("  Se despliega una tabla de conmutados básicos,\n");
lx[0]=radix;
dx[0]=0,dx[100]=1;
for(i=1;i<=101;i++)
   {
   lx[i]=radix-radix*qx[i][1];
   radix=radix-radix*qx[i][1];
   }
for(i=1;i<=99;i++)
   {
   dx[i]=lx[i]-lx[i+1];
   px[i]=lx[i+1]/lx[i];
   }
printf("\n  edad     lx        dx       qx        px \n\n");
for(i=0;i<=100;i++)
   {
   printf("%3d      %9.0f   %9.0f   %9.6f    %9.6f \n",i,lx[i],dx[i],qx[i][1],px[i]);
   if(lineas==14)
       {
         printf("\n\n Oprime una tecla para continuar...\n\n");
         getch();
         lineas=1;
       }
   lineas++;
   }
return 0;  }
```

6.9. Probabilidad y Estadística.

Programa 62

```
/* Ordenaciones con repeticion*/

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

int main(void)
{
 int obj,grupo,res;

 printf("\n  Obtiene el numero de observaciones con repetición de n objetos\n");
 printf("  tomados de m en m \n");
 printf("\n  Dame el numero de objetos  ");
 scanf("%d",&obj);
 printf("  De cuantos en cuantos quieres ordenarlos  ");
 scanf("%d",&grupo);
 res=pow(obj,grupo);
 printf("\n  Puedes ordenarlos de %d formas diferentes",res);
 return(0);
 }
```

Programa 63

```
/*Obtiene las Combinaciones de n elementos tomados de m en m*/

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

double fact(int);

int main(void)
{
 int n,m;
 double res;

 printf("\n  Este programa obtiene las Combinaciones de n elementos\n");
 printf("  tomados de m en m.\n");
 printf("\n  Dame el valor de n:  ");
 scanf("%d",&n);
 printf("  Dame el valor de m:  ");
 scanf("%d",&m);
 if( m>n)
   printf("\n  Error en los Datos\n  m debe ser mayor o igual a n");
 else
 {
   res= fact(n)/(fact(m)*fact(n-m));
   printf("\n  Las Combinaciones de %d en %d son: %.0lf",n,m,res);
 }
 return(0);
 }

double fact(int n)
{
int i;
double  r;

r=1;
for (i=1;i<n+1;i++)
   {
   r=r*i;
   }
return(r);
}
```

Programa 64

```
/* Càlculo de la media de n números con n<=100 */

# include <stdio.h>

void main(void)
{
  int  i,n;
  float ejemplo[100],media;

  printf("\n  Este programa calcula la media de n números con n<=100");
  printf("\n\n  A cuantos números le quieres sacar la media:  ");
  scanf("%d",&n);
  printf("\n");
  for (i=0; i<n; i++)
  {
    printf("  Escribe el numero %d : ", i+1);
    scanf("%f", &ejemplo[i]);
  }
  media=0;

  /* Ahora, se suman los números */

  for (i=0; i<10; i++)
    media=media+ejemplo[i];
  printf("\n  La media es %.2f \n", media/n);
}
```

Programa 65

```
/*Este programa recoge n datos (n<31),los ordena y nos dice con que frecuencia
se repite cada uno de ellos */

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

void main (void)
{

int datos[29],orden[29],f,i,n,aux,s=0,m;

clrscr();
printf("\n  Este programa recoge n datos (n<31),los ordena y nos");
printf("\n  dice con que frecuencia se repite cada uno de ellos \n\n");
printf("  A cuantos datos le quieres sacar la frecuencia, a lo mas 30:  ");
scanf("%d",&f );
printf("\n");
if (f>30 | f<=0)
  printf("\n El numero de datos es incorrecto \n");
else
{
   for(i=0;i<f;i++)
   {
     printf("  Dame el  %d dato ",i+1);
     scanf("%d",& datos[i]);
   }
for(n=0;n<f;n++)
  {
  for(i=0;i<f-1 ;i++)
    {
    if(datos[i]>datos[i+1])
        {
        aux=datos[i];
        datos[i]=datos[i+1];
        datos[i+1]=aux;
        }
    }
  }
for(i=0;i<f;i++)
  orden[i]=1;

printf("\n  TENGO LOS SIG NUMEROS  ");
for(i=0;i<f;i++)
```

```
  if(datos[i]==datos[i+1])
    orden[s]=orden[s]+1;
  else
    {
    printf("%d ",datos[i]);
    s=s+1;
    }
printf("\n\n CON FRECUENCIA          ");
for(i=0;i<s;i++)
      printf("%d ",orden[i]);
}

getche();
}
```

Programa 66

```
/* Este programa captura una encuesta y da algunos indicativos estadisticos*/

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

int menu(int n,int sexo[100],int estado[100],int nivel[100],int edad[100]);
int TABLA(int n,int datos[100],int vane);
int MODA(int n,int datos[100],int vane);
int MEDIANA(int n,int datos[100],int vane);
int FRECUENCIA(int n, int datos[100],int vane);
int MEDIA(int n, int datos[100]);
int MAXMIN(int n,int edad[],int vane);


int main()
{

int n=0,i,nivel[100]={0},sexo[100]={0},estado[100]={0},edad[100]={0};
char opcion;
clrscr();
printf("\t\t\" Programa para Capturar Encuestas \"\n\n");
printf("  Cuantas encuestas deseas capturar?  ");
scanf("%d",&n);

if(n>0)
{
  printf("  Responder colocando el numero:\n\n");

  for(i=0;i<n;i++)
  {
    printf("  Encuesta %d:\n",i+1);
    printf("\n  Sexo:\n\tFEMENINO   0\n\tMASCULINO  1\n  ");
    scanf("%d",&sexo[i]);

    if(sexo[i]>1)
    {
       printf("\n  0 equivale a Femenino\n  1 equivalea Masculino\n  Elija
              nuevamente:  ");
       scanf("%d",&sexo[i]);
    }
    printf("\n  Estado Civil:\n\tSOLTERO 0\n\tCASADO  1\n\tOTRO   2\n  ");
    scanf("%d",&estado[i]);

    if(estado[i]>2)
```

```
    {
       printf("\n  0 equivale a SOLTERO\n  1 equivalea CASADO\n");
       printf("  2 equivale OTRO\n  Elija nuevamente:  ");
       scanf("%d",&estado[i]);
    }
    printf("\n  Nivel de Estudios:\n\tBASICO        0\n\tTÉCNICO
1\n\tPROFESIONAL  2\n  ");
    scanf("%d",&nivel[i]);

    if(nivel[i]>2)
    {
       printf("\n  0 equivale a BASICO\n  1 equivalea TECNICO\n");
       printf("  2 equivale PROFESIONAL\n  Elija nuevamente:  ");
       scanf("%d",&nivel[i]);
    }

    printf("\n  EDAD:\n\t0.-  DE 0 a 20\n\t1.-  DE 21 a 40\n\t2.-  DE 41 a MAS\n  ");
    scanf("%d",&edad[i]);
    if(edad[i]>2)
    {
        printf("\n  0 equivale de 0-20\n  1 equivale de 21-40\n");
        printf("  2 equivale de 41-MAS\n  Elija nuevamente:  ");
        scanf("%d",&edad[i]);
    }


                }
                menu(n,sexo,estado,nivel,edad);
        }
        else
        {
                printf("Opcion No valida\n\tfin...");
                getch();
  }
  return 0;
}

int menu(int n,int sexo[],int estado[],int nivel[],int edad[])
{

int salir;
char o=0;
do{
  clrscr();
  printf("\t\tMENU\n\n");
  printf("  Opciones:(Elige por letra mayuscula)\n");
```

```
printf("\n  1. Sexo\n   (tabla de frecuencias)\n");
printf("\n  2. estado Civil:\n    (tabla de frecuencias y moda)\n");
printf("\n  3. niVel de estudios:\n    (tabla de frecuencias y mediana)\n");
printf("\n  4. eDad:\n    (tabla de frecuencias, media aritmética, moda, máximo y
      mínimo)\n");
printf("\n  5. todas las Anteriores");
printf("\n\n  6. saLir");
printf("\n\n\t\topcion:");
o=toupper(getche());
switch(o)
{
  case 'S':
  {
      clrscr();
      printf("  Valores para Sexo:\n");
      TABLA(n,sexo,2);
      break;
  }

  case 'C':
  {
      clrscr();
      printf("\n  Valores para Estado Civil:\n\n");
      printf("  0 SOLTERO\n  1 CASADO\n  2 OTRO\n");
      TABLA(n,estado,3);
      printf("\n\n\n  Moda: %d con %d
frecuencia(s)\n",MODA(n,estado,1),MODA(n,estado,0));
      getch();
      break;
  }

  case 'V':
  {
      clrscr();
      printf("\n  Valores para Nivel de Estudios:\n\n");
      printf("  0  BASICO\n  1 TECNICO\n  2 PROFESIONAL\n");
      TABLA(n,nivel,3);
      printf("\n\n\n  Mediana: %d con %d frecuencias\n", MEDIANA(n,nivel,1),
            MEDIANA(n,nivel,0));
      getch();
      break;
  }

  case 'D':
  {
      clrscr();
      printf("\n  Valores para Edad:\n\n");
```

```
        printf("  0  de 0-20\n  1 de 21-40\n  2 de 41-MAS\n");
        TABLA(n,edad,3);
        printf("\n\n\n  MODA: %d con %d frecuencias",MODA(n,edad,1),
               MODA(n,edad,0));
        printf("\n  MEDIA: %d ",MEDIA(n,edad));
        printf("\n  MINIMO: %d\n  MAXIMO: %d\n",MAXMIN(n,edad,1),
               MAXMIN(n,edad,0));
        getch();
        break;
    }

    case 'A':
    {
        clrscr();
        printf("\n  Valores para Sexo:\n");
        TABLA(n,sexo,2);
        clrscr();
        printf("\n  Valores para Estado Civil:\n\n");
        printf("  0  SOLTERO\n  1  CASADO\n  2  OTRO\n");
        TABLA(n,estado,3);
        printf("\n\n\n  Moda: %d con %d frecuencias\n",MODA(n,estado,1),
               MODA(n,estado,0));
        getch();
        clrscr();
        printf("\n  Valores para Nivel de Estudios:\n\n");
        printf("  0  BASICO\n  1 TECNICO\n  2 PROFESIONAL\n");
        TABLA(n,nivel,3);
        printf("\n\n\n  Mediana: %d con %d frecuencias\n",MEDIANA(n,nivel,1),
               MEDIANA(n,nivel,0));
        getch();
        clrscr();
        printf("\n  Valores para Edad:\n\n");
        printf("  0  de 0-20\n  1 de 21-40\n  2 de 41-MAS\n");
        TABLA(n,edad,3);
        printf("\n\n\n  Moda: %d con %d frecuencias",MODA(n,estado,1),
               MODA(n,estado,0));
        printf("\n  MEDIA: %d ",MEDIA(n,edad));
        printf("\n  MAXIMO: %d\n  MIMIMO
%d\n",MAXMIN(n,edad,1),MAXMIN(n,edad,0));
        getch();
        break;
    }

    case 'L':
    {
        printf("\n\t\nFin de programa... \n");
        getch();
```

```
        salir=1;
        break;
   }
}

}
while(salir!=1);
  return 0;
}

/*----------------------------------------*/

int TABLA(int n,int datos[],int vane)
{

int dato1=0,dato2=0,dato3=0;
int frec[3];

if(vane==2)
{
  int i,feme=0,mascu=0;

  for(i=0;i<n;i++)
  {
    if(datos[i]==0)
       feme=feme+1;
    else
       mascu++;
  }
  printf("\n\t\t\tTABLA DE FRECUENCIAS\n\n");
  printf("                        Frecuencia  Frecuencia     Frecuencia  \n");
  printf(" MarcaClase  Frecuencia  Relativa  Acumulada  AcumuladaRelativa\n");
  printf("\n  0 femenino      %d\t    %f\t   %d\t\t%f\n",
         feme,(float)feme/(float)n,feme,(float)feme/(float)n);
  printf("  1 masculino     %d\t    %f\t   %d\t\t%f",

mascu,(float)mascu/(float)n,mascu+feme,((float)mascu+(float)feme)/(float)n);
  getch();
}

else
{
  dato1=FRECUENCIA(n,datos,1);
  dato2=FRECUENCIA(n,datos,2);
  dato3=FRECUENCIA(n,datos,3);
  printf("\n\t\t\tTABLA DE FRECUENCIAS\n\n");
  printf("                        Frecuencia  Frecuencia     Frecuencia  \n");
```

```
  printf(" MarcaClase  Frecuencia  Relativa  Acumulada AcumuladaRelativa\n");
  printf("\n    0\t\t  %d\t    %f\t   %d\t\t%f",dato1,(float)dato1/(float)n,
       dato1,(float)dato1/(float)n);
  printf("\n    1\t\t  %d\t    %f\t   %d\t\t%f",dato2,(float)dato2/(float)n,dato1+dato2,
       ((float)dato1+(float)dato2)/(float)n);
  printf("\n    2\t\t  %d\t    %f\t   %d\t\t%f",dato3,(float)dato3/(float)n,
       dato1+dato2+dato3,((float)dato1+(float)dato2+(float)dato3)/(float)n);
}
return 0;
}


/*------------------------------------*/

int MODA(int n,int datos[],int vane)
{

int moda=0,a=0,dato1,dato2,dato3;
dato1=0,dato2=0,dato3=0;
dato1=FRECUENCIA(n,datos,1);
dato2=FRECUENCIA(n,datos,2);
dato3=FRECUENCIA(n,datos,3);
moda=dato1;
a=0;

if(dato2>dato1)
{
  moda=dato2;
  a=1;
}

if(dato3>dato2)
{
  moda=dato3;
  a=2;
}

if(vane==0) return moda;
if(vane==1) return a;
  return 0;

}
/*------------------------------------*/


int MEDIANA(int n,int datos[],int vane)
```

```
{

int mediana=0,b=0,dato1,dato2,dato3;
dato1=0,dato2=0,dato3=0;
dato1=FRECUENCIA(n,datos,1);
dato2=FRECUENCIA(n,datos,2);
dato3=FRECUENCIA(n,datos,3);
mediana=dato1;
b=0;

if(dato2>dato1)
{
  if(dato3>dato2)
  {
    mediana=dato2;
    b=1;
  }
}

if(dato3>dato1)
{
  if(dato2>dato3)
  {
    mediana=dato3;
    b=2;
  }
}

  if(vane==0) return mediana;
  if(vane==1) return b;
    return 0;

}

/*---------------------------------*/

int FRECUENCIA(int n,int datos[],int vane)
{

int dato1=0,dato2=0,dato3=0,i;

for(i=0;i<n;i++)
{
  if(datos[i]==0)
    dato1++;
  if(datos[i]==1)
    dato2++;
```

```
  if(datos[i]==2)
    dato3++;
}

if(vane==1)
  return dato1;
if(vane==2)
  return dato2;
if(vane==3)
  return dato3;
return 0;

}

/*--------------------------------------*/

int MEDIA(int n, int datos[])
{

int media=0,dato1,dato2,dato3;
dato1=0,dato2=0,dato3=0;

dato1=FRECUENCIA(n,datos,1);
dato2=FRECUENCIA(n,datos,2);
dato3=FRECUENCIA(n,datos,3);

media=( 10*dato1+30*dato2+50*dato3)/3;
return (media);
}

/*--------------------------------------*/

int MAXMIN(int n,int edad[],int vane)
{

int mami[3],frec[3],max=0,min=0,dato1,dato2,dato3;
dato1=0,dato2=0,dato3=0;

dato1=FRECUENCIA(n,edad,1);
dato2=FRECUENCIA(n,edad,2);
dato3=FRECUENCIA(n,edad,3);

if(dato1>0)
  min=10;
else
  {
  if(dato2>0)
```

```
      min=30;
    else
      min=50;
    }

if(dato3>0)
   max=50;
else
{
   if(dato2>0)
     max=30;
   else
     max=10;
}

if(vane==0)
   return max;
if(vane==1)
   return min;
return 0;
}
```

Conclusiones.

- ✓ La programación es una herramienta que podemos explotar mucho los Actuarios; porque una vez que aprendemos un lenguaje de programación y combinamos éste con el potencial matemático que tenemos por nuestra formación académica, somos capaces de crear desde los más sencillos programas hasta los mas complejos sistemas.
- ✓ Se puede crear todavía mucho software actuarial que nos permita ver resultados concretos y nos evite la talacha innecesaria.
- ✓ Se puede programar para varias de las materias que se incluyen en los planes de estudio de la carrera de Actuaría, materias como: Álgebra, Geometría, Matemáticas Financieras, Cálculo Integral, Análisis Numérico, Análisis Matemático, Cálculo Actuarial, Probabilidad, Estadística, etc.
- ✓ Programar nos permite crear y personalizar nuestro propio software.

Apéndice 1

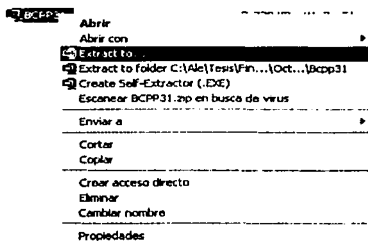
Cómo usar un Compilador de C

He elaborado este pequeño apéndice para mostrar desde cómo se instala un compilador de C hasta algunas de las cosas más básicas que uno tiene que saber respecto a su uso.

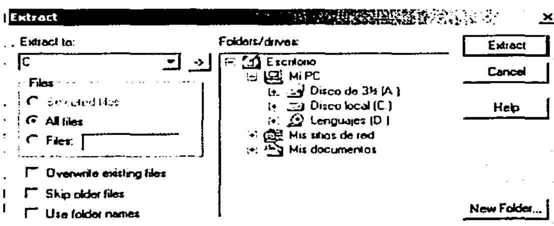
Ejemplificaré lo anterior con un compilador para C que incluyo en este trabajo. Como es de esperarse, las instalaciones de los demás compiladores son similares.

Para instalar el compilador hay que seguir los siguientes pasos:

- ✓ Desempaquetar el archivo BCPP31.ZIP en un directorio. Para este caso lo haré en el directorio raíz c:\ , lo cual se hace de la siguiente manera:
- ✓ Nos colocamos en el archivo y damos un clic con el botón derecho, a continuación nos vamos a "Extract to" y pulsamos un "Enter"

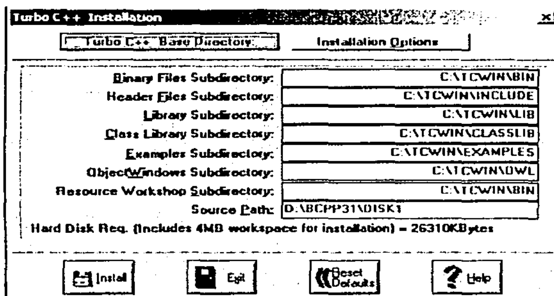


- ✓ En la siguiente pantalla pulsamos "I Agree" para ver lo siguiente:



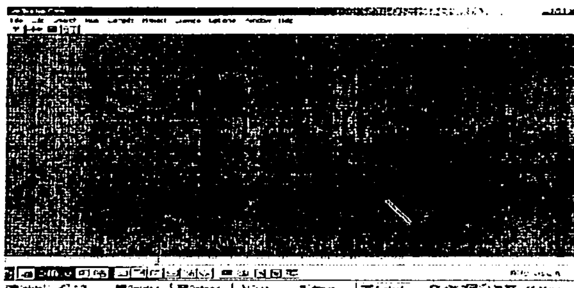
En donde colocamos **c:** en Extrac to

✓ Una vez desempaquetado nos vamos al directorio c:\disk1 en donde damos un enter al archivo INSTALL y nos aparece la siguiente pantalla

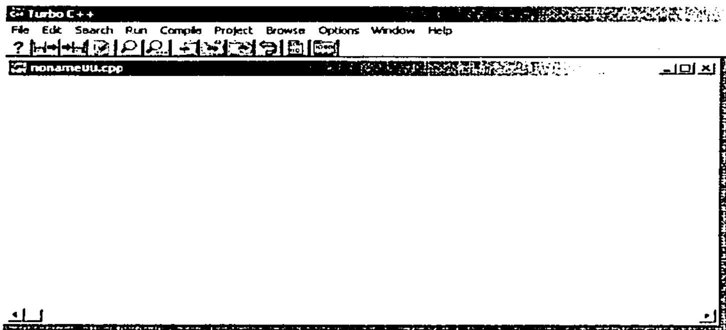


donde pulsamos "Install" y automáticamente se comienza a instalar el compilador. Al terminar nos aparecerán pantallas donde debemos pulsar OK y listo. El compilador ha sido instalado.

Una vez instalado éste, cuando accedemos a él, lo primero que nos encontramos es la siguiente pantalla.



Para comenzar a escribir un programa nos vamos a la opción FILE posteriormente nos colocamos en NEW y nos aparece una pantalla como la siguiente.



La cual nos muestra un editor de texto en donde podemos comenzar a escribir nuestro código del programa.

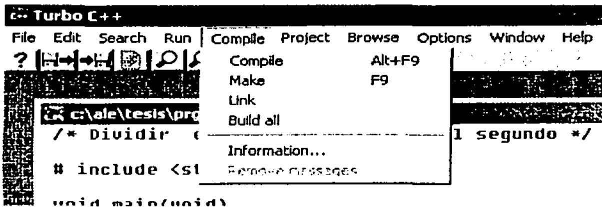
Las opciones de Guardar son similares a las de Windows, pero hay que poner mucha atención a la hora de guardar el archivo sobre todo en los siguientes puntos.

- ✓ Debe guardarse el archivo con una extensión **.c**
- ✓ Debe indicarse la ruta empezando por la unidad en que se encuentra, seguido de cada uno de los directorios y subdirectorios correspondientes; por ejemplo:

  C:\ale\progc\programa1.c

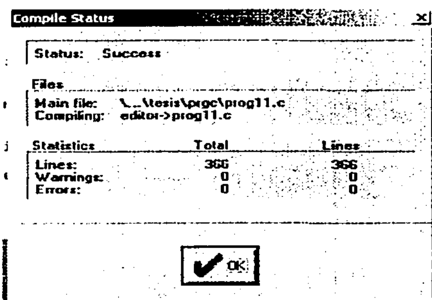
Si queremos abrir un archivo; lo hacemos de la misma manera que utilizamos para guardarlo especificando toda la ruta del archivo.

Para compilar nuestro programa y verificar que no tiene errores



nos vamos a la opción COMPILE, depués otra vez COMPILE o simplemente pulsamos Alt+F9.

Donde nos aparece una pantalla como la siguiente:

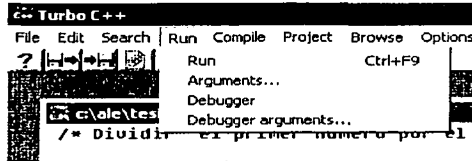


Aquí es importante observar que se tengan los Warnings y los Errors en 0.

Un Warning nos previene de un posible error que estamos cometiendo mientras que un error no nos dejara correr el problema hasta que se corrija.

Para correr un programa es necesario no tener Errors y lo deseable es no tener Warnings pero se puede dar el caso de que con algunos Warnings el programa corra.

Para correr un programa hacemos lo siguiente:



Nos vamos a la opción RUN, posteriormente RUN o simplemente tecleamos Ctrl+F9.

A continuación se despliega una pantalla como la que has visto en los ejemplos de este trabajo.

Existen muchísimas opciones más; pero para poder comenzar a trabajar con este compilador las anteriores son más que suficientes. Si deseas saber un poquito más sobre el uso de algunas de las opciones del Menú puedes hacer uso de la Ayuda pulsando F1 o accediendo por medio del menú con el comando "help".

Apéndice 2

Programas Incluidos.

Número de Programa	Qué nos ejemplifica		Qué hace
	Aplicación Actuarial	Programación en C	
1		printf ()	Imprime en la pantalla el tamaño de bytes en los diferentes tipos de datos en C
2		Especificadores de Formato	Nos da diferentes salidas de datos mediante el uso de los especificadores de formato
3		getch(), getche() y getchar()	Ilustra el uso de getch(), getche() y getchar()
4		puts(), gets(), printf() y scanf()	Despierta en el lector la curiosidad de encontrar las diferencias entre el uso de puts y printf, y de gets y scanf
5		Variables locales	Muestra el uso de una variable declarada en un bloque especifico, siendo el caso de que existe otra declarada al inicio del programa
6		Constantes de Carácter de Barra Invertida	Exhibe cómo imprimir algunos caracteres que no se pueden introducir mediante el teclado
7		Asignaciones compuestas	Nos ejemplifica cómo utilizar una asignación compuesta
8		Operadores ++, --	Nos enseña el uso correcto de los operadores de incremento y decremento; así como, cuando usar uno y cuando el otro
9		Operadores de apuntadores	Este programa realiza el paso de un valor de una variable a otra utilizando una variable apuntador
10		If()	Si un número introducido mediante el tecleado se encuentra entre 1 y 10 lo eleva a la cuarta potencia
11		if ()– else	Dados dos números divide el primero entre el segundo siempre y cuando el segundo sea diferente de cero
12		if's anidados	Dados tres números diferentes nos muestra el menor

Programas Incluidos.

Número de Programa	Qué nos ejemplifica		Qué hace
	Aplicación Actuarial	Programación en C	
13		Switch	Hace la conversión de números entre las base decimal, octal y hexadecimal dadas 4 opciones diferentes
14		switch anidados	Realiza una sencilla base de datos de vendedores por regiones
15		For	Suma n números
16		For	Este programa imprime en la pantalla los números pares del 0 al 100.
17		While	Despliega los múltiplos del 4 entre 0 y 40
18	Álgebra Superior	While	Calcula el mcd de 2 números
19		While	Nos calcula algunos números que al cuadrado pueden expresarse como la suma de otros 2 cuadrados
20		do - while	Pide un número y mientras este sea menor que 100 se ejecutara el bloque
21		do - while	Se asegura que el usuario especifique la opción valida
22		do - while	Imprime un menú mientras el usuario así lo desee
23		Break	Se interrumpe un ciclo que imprimiría los números del 1 al 100
24		exit( )	Sale del programa al encontrar la instrucción exit ()
25		Continue	Cuenta los espacios en blanco de una cadena proporcionada por el usuario
26		Cadenas	Muestra el uso de continue con el ciclo while
27		Cadenas	Copia una cadena a una variable
28		Cadenas	Concatena el contenido de 2 cadenas
29		Cadenas	Compara dos cadenas y nos regresa el valor de la comparación
30		Cadenas	Dada una cadena nos dice cuántos caracteres tiene

Programas Incluidos.

Número de Programa	Que nos ejemplifica		Qué hace
	Aplicación Actuarial	Programación en C	
31		Arreglos	Llena un arreglo y nos despliega su contenido
32		Arreglos	Crea una base de datos de 10 cadenas
33		Apuntadores	Nos muestra el uso de los operadores apuntador, los modificadores que deben de utilizarse con printf() para generar la salida correcta
34		Apuntadores	ejemplifica el uso de apuntadores
35		Apuntadores	ejemplifica el uso de apuntadores
36		Arreglos	Accedemos a los elementos de un arreglo mediante la indexación de los mismos
37		arreglos	Accedemos a los elementos de un arreglo mediante la aritmética de apuntadores
38		in dirección múltiple	Realiza una indirección múltiple e imprime su valor
39		funciones	Llama una función la cual contiene 2 parámetros que el usuario introduce mediante el teclado
40		funciones	Crea una llamada por valor a una función
41		funciones	Crea una llamada por referencia a una función
42		funciones	Pasa un arreglo como argumento de una función el cual es modificado dentro de la misma
43		funciones	Pasa un arreglo como argumento de una función el cual **no** es modificado dentro de la misma
44		funciones	Nos enseña a regresar de una función
45		funciones	Nos muestra cómo regresar el valor de una función
46		funciones	La devolución de un apuntador de una función

Programas Incluidos.

Número de Programa	Que nos ejemplifica		Qué hace
	Aplicación Actuarial	Programación en C	
47	Álgebra Superior		Calcula las raíces mediante la formula de la ecuación de segundo grado
48	Álgebra Superior		Calcula un determinante
49	Álgebra Superior		Calcula c a la n con c un número real y n un entero
50	Álgebra Superior		Calcula n!
51	Álgebra Lineal		Realiza la suma de 2 matrices
52	Álgebra Lineal		Producto de 2 matrices de a lo más 10x10
53	Geometria Analitica		Calcula el producto punto de 2 vectores
54	Geometria Analitica		Obtiene la norma de un vector
55	Matemáticas Financieras		Calcula un monto utilizando interés compuesto
56	Matemáticas Financieras		Calcula el Monto y Valor Presente de una Anualidad
57	Cálculo integral		Calcula las sumas inferiores de Riemman para la X^2
58	Análisis Numérico		Obtiene la raiz cuadrada mediante aproximaciones
59	Análisis Numérico		Calcula la raiz cuadrada por el Método Newton
60	Análisis Matemático		Realiza la convergencia de la Serie Updown dado un número inicial
61	Calculo Actuarial		Calcula algunos Conmutados Básicos
62	Probabilidad		Calcula las ordenaciones con repetición
63	Probabilidad		Obtiene las Combinaciones
64	Estadística		Calcula la media de n números
65	Estadística		Obtiene la frecuencia de 2 números
66	Estadística		Captura un encuesta y nos da algunos indicadores estadísticos

Referencias

Bibliografía

Anderson, Paul y Anderson Gail. **Advanced C Tips and Techiques.**
Hayden Books.- EEUU, 1988. 446p.

Duglas, Hergert. **The ABC's of QuickC.**
SYBEX.- EEUU, 1989. 300p.

Brian, Kernigham y Dennis Ritchie. **El lenguaje de programación C.**
Prentice Hall.- Englewood Cliffs, N.J, 1991. 999p

Schildt, Herbert. **Manual de Referencia C.**
McGraw-Hill.- España, 2001. 999p.

Kernighan, Brian W. **El Lenguaje de Programación C.**
Prentice Hall Hispanamericana.- México, 1985. 225p.

Kelley, Al. **Lenguaje C: Introducción a la Programaciòn**
Âddison.Wesley Iberoamericana.- México, 1987. 392p.

Hancock, Les. **Introducción al Lenguaje C.**
Byte Books.- Madrid, 1988. 305p.

Kochan, Stephen. **Programming in C.**
Hayden.- Indianapolis, Indiana, 1988. 467p.

Paginas de los Planes de Estudio

www.unam.com.mx

www.itam.mx

www.gdl.uag.mx

www.uas.mx

www.pue.udla.mx

Paginas de Referencia de C

Curso de C
http://members.es.tripod.de/ncabanes/c1.htm

Curso de Introducción al Lenguaje C
http://webpages.ull.es/users/fsande/talf/cursoc/node1.htm

Lenguaje C
http://www.cienciasmisticas.com.ar/infor/c.html

Aprendiendo C.
http://150.186.93.70/~joseleon/

Con Clase
http://www.conclase.net/

C World
http://www.ciudadfutura.com/cworld/cursodec.htm

El Rincón de C
http://www.elrincondelc.com/

El Lenguaje C
http://www.iespana.es/mundolinux/progr/manc/lengc.htm

Introducción al Lenguaje C
http://labsopa.dis.ulpgc.es/cpp/intro_c/

Programación en C
http://cfp401.freeservers.com/cursos/c1/c.htm

El Rincón del Programador
http://personales.jet.es/perell/index.html

C-Almohadilla
http://www.c-almohadilla.es.vg/

Lenguaje de Programación en C
http://www.lafacu.com/apuntes/informatica/lenguajec_1/default.htm

Lenguaje C
http://www.jeanpaul.com.ar/

Curso Básico de Lenguaje C
http://webdia.cem.itesm.mx/ac/rogomez/lengC.html

Lenguaje C
http://www.unixsup.com/cursos/lenguajeC.html