24
2ej

Universidad Nacional Autónoma de México

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

ARAGON

INGENIERIA EN COMPUTACION

APUNTES PARA LA ASIGNATURA PROGRAMACION ESTRUCTURADA Y CARACTERISTICAS DE LENGUAJE

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A

ERNESTO PEÑALOZA ROMERO

ENEP

ARAGON

MEXICO, D.F. 1993.

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INTRODUCCION

Dentro del plan de estudios de la carrera de Ingeniería en computación que se imparte en la Escuela Nacional de Estudios Profesionales Plantel Aragón, de la Universidad Nacional Autónoma de México se encuentra la materia de *Programación Estructurada y Características de Lenguaje*. La materia fue diseñada originalmente para mostrar al estudiante las técnicas planteadas por la filosofía de la Programación Estructurada y mostrar las diversas características que mostraban los lenguajes de programación. Con la renovación de la currícula de la carrera, los objetivos han cambiado hacia rumbos más modernos dentro de un área de activa creación y desarrollo técnológico.

La programación ha sido un área que ha ocupado a más de una persona en estudios sobre eficiencia y productividad. Las muy diversas escuelas que han surgido y el cambio en los paradigmas de la programación, hacen necesaria una revaloración más amplia de lo que la computación y la programación son para la currícula de nuestros estudiantes de computación.

Los presentes apuntes no pretenden mostrar una serie de técnicas que se deben seguir como receta de cocina. Su fin es formar una consciencia más científica en la mente del estudiante sobre los enfoques existentes. No es posible plantear filosofías consideradas como mutuamente excluyentes (estructurada vs objetos) sin establecer la premisa de que estas escuelas de pensamiento son el fruto de consecutivas evoluciones en el pensamiento de los profesionales de la programación. Es más, deben de ser planteadas como secuencias y consecuencias de un mismo fin.

Programar ya no es tan sólo un arte y es absolutamente necesario tomar consciencia que ésta disciplina tiene necesidad de la aplicación de uno o más métodos para el diseño de los sistemas que la sociedad requiere.

Por otra parte, la currícula de la carrera deja de lado temas considerados como *teóricos* que son de importancia fundamental para la cabal comprensión de los conceptos que son presentados en el temario de la materia. La máquina de Turing y la computabilidad son conceptos fundamentales para cualquier persona que decida plantear un problema en forma algorítmica. Dado que se ha considerado que estos temas son fudamento de diversos conceptos de la programación estructurada y de la programación orientada a objetos, se han presentado sin perder de vista los objetivos de la materia.

El capítulo uno introduce al alumno en el concepto de abstracción que servirá de piedra fundamental en el desarrollo del diseño por pseudocódigo y más adelante de la programación orientada a los objetos. Se hace incapié en los requisitos deseables de los módulos y se muestran las estructuras de control así como la necesidad de usar solamente dichas estructuras en la construcción de los programas.

El capítulo dos presentará al estudiante el lenguaje C. Sin embargo, el lenguaje C es en realidad un pretexto para mostrar las técnicas estructuradas discutidas en el capítulo anterior. El capítulo no está orientado sólo a la enseñanza de un lenguaje, sino a la enseñanza de un buen estilo de programación. Y el buen estilo siempre será independiente del lenguaje que se trabaje.

Para lograr un buen estilo se muestran programas completos. El pseudocódigo se muestra como un auxiliar en la comprensión de aquellos programas cuya complejidad sea evidente para el alumno. A su vez, se han tratado de describir las características de C basándose en el estándar ANSI

pero sin dejar de mencionar las diferencias con versiones avanzadas de C++ y algunas de las diferencias con el C de Ritchie. Esto asegurará al alumno la portabilidad de sus programas cuando emigre del equipo multiusuario actualmente utilizado en la ENEP hacia la ominipresente PC.

El capítulo tres será una continuación de los conceptos vertidos en el capítulo uno. Personalmete estoy convencido de que ambas filosofías no se contraponen, sino que son complementarias. El enfoque que se le da a la programación orientada a los objetos proviene de los fundamentos de la programación estructurada.

La presente obra no pretende ser un tratado ni un libro de texto completo y suficiente. La amplitud de los temas ha impedido tratarlos con la profundidad deseada. Sin embargo, constituyen en sí mismos valiosos auxiliares que presentan una visión integral de las dos principales corrientes de programación que existen en este momento en el mundo de la programación.

Se ha cuidado la tipografía de la obra para resaltar los puntos importantes. Cuando se presenta un término nuevo se han escrito con letra cursiva *Times New Roman*. Las funciones se han representado colocando el nombre de la misma y un par de paréntesis. Los listados de los programas se han escrito con letra `courier new` y en negrita. Algunos conceptos importantes y nombres de variables se han escrito con letra **arial** en negrita. Por último, las entradas a los programas se han colocado en letra universe normal.

Por otra parte, se ha tenido en mente la heterogeneidad en los conocimientos con los que los alumnos llegan a esta materia, por lo que se ha incluído un glosario de términos que el alumno puede consultar cada vez que el texto cite un nuevo concepto o que éste no sea muy claro para el alumno. En general el texto es sencillo; se maneja la idea de un conocimiento intuitivo de ciertos conceptos por parte del alumno

Esperando que estos apuntes sean verdadera ayuda para todos los lectores, iniciados o avanzados, se agradecerá cualquier sugerencia o corrección que se haga llegar a la Coordinación de Ingeniería en computación de la ENEP Aragón o al Apartado Postal 61-014, en la colonia Juarez. CP 06601.

Ernesto Peñaloza Romero

I N D I C E

METODOLOGIA DE LA PROGRAMACION ESTRUCTURADA

LENGUAJE C

LA METODOLOGÍA DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

JUSTIFICACION Y ALCANCE

La mayoría de los cursos para aprender a programar, son esencialmente iguales a los cursos de manejo, en los cuales enseñan a conducir el auto, pero no la manera de usarlo para llegar a nuestro destino.

Edsger W Dijkstra

CAPITULO

I

METODOLOGIA DE LA PROGRAMACION ESTRUCTURADA

En los 70's, todos estaban a su favor, cada fabricante promovía productos soportándola, cada administrador de sistemas pagaba por servirla, cada programador la practicaba (de diferente manera). Y nadie sabía realmente que era la programación estructurada.

Rentsch,T

I.1 La programación estructurada

La historia del ser humano está estrechamente ligada a la historia del desarrollo de sus sistemas de comunicación de la información. Un mundo como el actual, no es posible desasociarlo de la revolución que las computadoras han acarreado a la vida cotidiana de millones de personas en todos los rincones del planeta.

Los sistemas de información controlan lo mismo un vuelo espacial que una compra sencilla en un supermercado. El mundo de fantasía que crean los efectos especiales o la música sintética del *New Edge* son producto también de esa máquina que comenzó como una idea en la mente de los pioneros de la computación.

El desarrollo de los actuales sistemas de información ha sido posible gracias al desarrollo paralelo de un conjunto de técnicas y procedimientos para la organización del trabajo de creación del software, y dicho desarrollo está estrechamente unido a los avances tecnológicos de hardware. Estas técnicas han venido dándose como productos necesarios de una sociedad consumidora de información. *Más poder tiene quien más velozmente puede disponer de información y sobre esta base decidir las acciones que conviene tomar.* Las grandes empresas de todo el orbe gastan millones de dólares en el desarrollo de sistemas de cómputo que resuelvan los problemas generados por el enorme volúmen de operaciones realizadas en un sólo dia de trabajo. El trabajo está a cargo de los nuevos ingenieros de sistemas y de computación que solucionan en forma cotidiana estos problemas con la ayuda de técnicas de desarrollo y mantenimiento de los programas que conforman a dichos sistemas de información.

EL PROGRAMA COMO UN ELEMENTO INTANGIBLE

Un programa no posee un color definido, un peso específico, un olor determinado o alguna característica física perfectamente mesurable a través de la cual nuestros sentidos la puedan percibir. Es más, un programa es una entidad dinámica que sólo existe como tal en el momento de ejecutarse. Antes y después de ello un programa es sólo un conjunto de instrucciones estáticas que no sirven en absoluto como no sea para ocupar espacio en memoria.

Un programa resulta por tanto, una entidad sumamente abstracta. Esta naturaleza hace difícil el establecimiento de algún tipo de norma para la definición de las características deseables en un programa o un sistema.

Con esto no se está menospreciando el programa. Después de todo constituyen la mitad de cualquier computadora útil. Precisamente esta importancia hace necesario el mirar más detenidamente a ese elemento al cual no se le puede medir ningún parámetro físico para establecer su correcto funcionamiento dentro de la computadora.

LA ABSTRACCION Y EL SOFTWARE

¿ Es la naturaleza abstracta de un programa una ventaja o una desventaja?

Un ente abstracto no puede ser cuantificado. ¿ Cómo puede decirse que un programa es 1.34 veces mejor que otro? ¿ Qué establece lo que es mejor para un programa?

La intangibilidad de las características de un buen programa parecen no dar cabida a mediciones de cualquier tipo. Sin embargo, ha sido la abstración la que sacó al ser humano de su estado cavernario y ha logrado crear la sociedad tecnificada que hoy en día poseemos. *La abstración es la clave del diseño de software.*

En los primeros tiempos de la computación, los programadores enviaban todas las instrucciones a las computadoras directamente por conmutadores binarios colocados en los paneles de control. Ese es el nivel más elemental en que se puede trabajar un computador digital. El siguiente paso fué la creación de los mnemónicos que evitaron que los programadores tuvieran que recordar las largas cadenas de bits que introducían constantes errores. Se abstrajo la función de las cadenas de bits y se les dió un nombre. *Por abstracción entendemos el proceso a través del cual se obtienen las características esenciales de un objeto determinado.*

Un paso más para lograr un plano de abstración superior, fué la habilidad para crear instrucciones definidas por el programador. A los conjuntos de instrucciones se les agrupaba nombrándolas. Así una sencilla macroinstrucción ejecuta muchas cosas a la vez.

Los lenguajes de alto nivel permitieron a los programadores evitar pensar en la arquitectura de la máquina en la cual se implantan sus sistemas, para dedicarse a pensar en la solución del problema. En general, *cualquier lenguaje posee cierto conjunto de estructuras de control suficientes que le permiten implantar cualquier problema que sea factible de expresar en forma algorítmica.* Esta definición abarca inclusive a los lenguajes no estructurados como FORTRAN o BASIC.

Las secuencias de instrucciones de alto nivel pueden ser agrupadas dentro de subprogramas y ser invocadas con una sola sentencia. Los programadores no necesitan conocer los detalles de la implantación. Unicamente necesitan conocer cuales son los insumos (entradas) del subprograma, cual es la función que éste realiza y cual es el resultado.

Un programa al ser una entidad abstracta por sí misma, nos permite evitar pensar en los detalles más relevantes y nos evita distraernos en detalles que carecen de importancia. Esto no deja de ser una trampa en sí misma, ya que en ocasiones las características fundamentales de un programa no son tan *visibles* como pudiera esperarse. Generalmente problemas frecuentes en los sistemas tales como caídas de sistema, error o pérdidas en la información, baja eficiencia, etc, son debidos a que los programas no fueron enfocados correctamente desde la etapa de diseño.

Si cualquier lenguaje de programación posee un conjunto mínimo y finito de instrucciones de control que podemos considerar como fundamentales, entonces cualquier programador, al saber programar con ese conjunto finito de instrucciones, podrá implantar cualquier algoritmo con total independencia del lenguaje en cuestión. Se logra la abstracción que nos permite desentendernos del lenguaje de implantación y de las diferencias entre estos, para poder enfocar nuestros pensamientos a la resolución de problemas.

La Programación Estructurada hace uso de la abstración al encontrar las formas más comunes de control de un programa (tal como una decisión **SI-ENTONCES**) que están incorporadas dentro de los lenguajes de programación. Esto nos permite abstraer el control, de los detalles del lenguaje, para lograr cambiar la secuencia de ejecución.

La intangibilidad de un programa se toma entonces en una ventaja.

PROGRAMADORES Y ARTE

La programación estructurada tuvo su mayor auge en la década de los 70's. Se dictaron múltiples conferencias para extender sus conceptos, su vendieron muchos sistemas diseñados bajo en enfoque modular. Han pasado muchos años desde su surgimiento, y a pesar de las ventajas de este tipo de programación, una gran cantidad de programadores (inclusive profesionales) aún no comprenden la modularidad o las estructuras de control.

Esto proviene del perfil con el cual se forma a los programadores. Se ha repetido en muchos libros que programar es un arte, esto convierte por ende al programador en un artista. Se nos ha educado para

pensar que un programa es toda una obra de arte nacida de la inspiración de los genios del teclado y el monitor, razón por la cual cada programador ve a su obra como algo irrepetible en la historia de la humanidad y que ,por tanto, está totalmente libre de un juicio de valor. Al ser los programas elevados al nivel de obras de arte, nadie puede emitir juicios axiológicos válidos sobre la excelencia de dicha obra.

Lo abstracto del término *programa* y la concepción artística totalmente subjetiva que hace el programador de su obra, limitan el criterio para aplicar una metodología a la creación del software.

¿ Deben dos analistas de sistemas desarrollar una lista idéntica de requerimientos cuando estudian en forma simultánea e independiente la misma situación? En forma intuitiva así debería de ser. ¿ Por qué entonces dos programadores no deberían de usar las mismas estructuras para implantar el mismo algoritmo?

Es válido y totalmente humano adjudicar a la creatividad de cada persona un papel preponderante durante la implantación de programas. Pero para lograr hacer de la computación una ciencia es preciso que los profesionales sigan una disciplina de diseño e implantación. Ya no es posible seguir pensando en que la programación es un arte. Los métodos de inteligencia artificial (AI), diseño asistido por computadora (CAD), sistemas expertos, etc. exigen algo más que *arte*. Los alquimistas no sobrevivieron a los químicos. Los artistas en programación no sobrevivirán a los ingenieros en software. La sociedad requiere de personas preparadas y, estando a punto de entrar al siglo XXI, no es posible que el método científico no llegue a un área que desde su nacimiento se ha considerado *cosa de sabios*.

¿ FILOSOFIA, METODO O TECNICAS?

Aquí se puede entrar en todo un debate. Las ciencias físicas o concretas tienen un conjunto de objetos de estudio que siguen reglas rígidas que se pueden comprobar y rebatir. Las ciencias humanísticas poseen también objetos de estudio tan reales como las ciencias concretas, pero no poseen métodos de medición y de comprobación de leyes tan precisos como las de las ciencias concretas. Esto las limita en cuanto a su viabilidad como ciencias predictivas. La falta de predictibilidad es lo que las pone en tela de juicio como ciencia para el grueso de la gente. ¿A cuál de los dos conjuntos pertenece la computación?. La ciencia posee un conjunto de axiomas en los cuales se fundamenta toda su estructura. ¿Cuáles son los soportes de la computación que la validen como una disciplina científica válida y predictible?

Hablar de ciencias de la computación es enfrentarse a los principios clásicos del método científico. Por ello se podría pensar que los métodos de los cuales se valen los profesionales de la computación no pasan de ser un conjunto de técnicas más o menos empíricas y sujetas a una apreciación un tanto cuanto subjetiva. Existen diversas escuelas que nos *enseñan "la mejor manera"* de implantar un sistema. ¿En donde quedó la unidad inherente a las ciencias de la computación si existen tantas escuelas distintas de pensamiento para un sólo objetivo que es el de crear sistemas ?

La Programación Estructurada es una filosofía para la implantación de algoritmos a través de un conjunto finito de estructuras bien organizadas. Es esta la manera como fué definida por Constantine y aún sigue siendo válido este enfoque.

Tal vez suene extraño hablar de filosofías dentro de un ámbito tan eminentemente técnico como lo es la ingeniería. Estamos más acostumbrados a pensar en función de modelos y procedimientos científicamente probados y comprobados para la solución de los problemas a los cuales nos enfrentamos en nuestra vida profesional. Esta forma de ver a las herramientas con las cuales trabaja el ingeniero hacen pensar que cualquier cosa que sea utilizable debe caer dentro de una especie de receta de cocina en la cual sólo es necesario seguir los pasos para obtener la solución deseada. Las técnicas son la herramienta básica de un ingeniero pero no lo constituyen en sí mismo. Tal y como comenta Boch:

> *El ingeniero de software principiante está siempre en busca de alguna fórmula mágica, alguna inovación tecnológica cuya aplicación inmediata le permitirá hacer fácil el proceso de desarrollo de software. Es propio de un ingeniero profesional en software saber que no existe*

aseveración conocida por la mayor parte de los programadores dice que no es posible probar un sistema exhaustivamente. Siempre quedará una opción sin probar, un caso en particular que provocará a la larga un error.

El grado de confiabilidad de un producto tiene una relación directa con el costo de una falla. No es lo mismo una falla que obstruye la terminal por dos minutos y que sólo produce una molestia pequeña a un usuario y una falla que hace explotar el cuarto reactor en una central nuclear.

Estas dos características son por sí mismas evidentes para el usuario. Al convivir con el sistema, el usuario sólo puede ver la pantalla y los resultados.

Desde un punto de vista más técnico tenemos la *legibilidad* y el *estilo de programación*. Un programa pasa la mayor parte de su vida en etapas de corrección de errores, expansiones y actualizaciones. Si ese programa no es claro y de fácil lectura, muy probablemente dificultará cualquier intento de modificación.

De hecho una gran parte de los programadores prefieren reescribir un programa que modificar uno ya hecho. La desorganización existente en los códigos, la mala o nula documentación, las malas prácticas de codificación, hacen que sea menos costoso reescribir código.

La *eficiencia* es un factor de calidad muy relativo. Son muchos los programadores que buscan la eficiencia por encima de la legibilidad, la claridad, la funcionalidad y ocasionalmente hasta por encima de la confiabilidad. El mismo término *eficiencia*, sin embargo, no es muy claro para ellos.

Se habla de eficiencia cuando se optimizan los recursos del computador. Si lo que hace falta es memoria principal, un sistema de paginación y/o segmentación es una solución eficiente aunque utilice mayor tiempo de U.C.P. Si a esto hay que aunar el manejo de gráficos, entonces es necesario replantear la solución para evitar una sobrecarga del procesador.

La eficiencia está en función del tipo de aplicación con la cual se está trabajando. Y sólo es realmente trascendente cuando su inexistencia afecte seriamente el desempeño del sistema en líneas generales.

En líneas generales estas son las pautas que deben ser consideradas cuando se evalúe la calidad de un programa. Obsérvese atentamente que no se está abogando por algún estilo de programación en particular. Las características que hacen a un buen programa son en una buena medida inherentes a su naturaleza. No es trascendente que un programa sea escrito bajo un enfoque estructurado u orientado a objetos o bajo cualquier otro enfoque.

Seguir una línea totalmente intuitiva de *"diseño"* provoca que los programas resulten difíciles de mantener, de expander, y de probar. El enfoque estructurado hace más barato el desarrollo de software al establecer un enfoque disciplinado que hace a los programas más sencillos de mantener, depurar y modificar.

1.1.1 Definición del programa

MODELADO DE LA REALIDAD

A lo largo de la historia del hombre, éste ha buscado maneras para representar el mundo que lo rodea. Los diversos niveles de abstracción definen en cierta forma el grado evolutivo del pensamiento.

Observar un problema y representarlo de alguna forma es un trabajo que suele resultar complicado. Si se trata de un sistema manual en el que estén involucrados elementos ajenos a los procedimientos, esto dificulta la representación del mismo. Por ejemplo, en un sistema de inventarios se tienen diversas entradas al almacén, salidas por ventas y traspasos o reposiciones. Las existencias físicas en el almacén son función de todas esas variables. En general un análisis se planteará en función a los parámetros más comunes dentro de la operación cotidiana. Un robo no es parte de la operación común del almacén, sin embargo es uno de los casos posibles que deben ser contemplados dentro de la representación de la operación de dicho almacén.

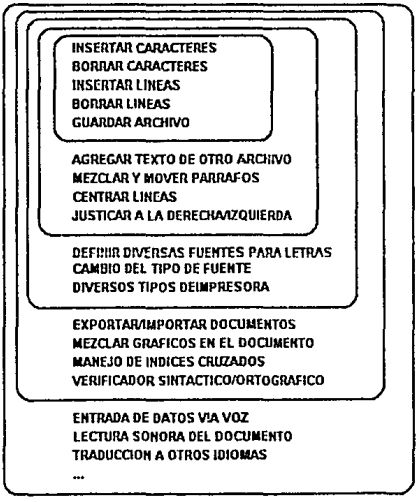
Un problema de *la vida real* y que debe ser resuelto por una computadora debe cumplir con un requisito: que sea *computable*. La computabilidad de un problema es la representación de dicho problema a través de proposiciones que lo resuelven. Es decir, para que un problema sea computable debe existir un conjunto de proposiciones que den solución a dicho problema.

Si un modelo representa correctamente a la realidad que simula, cualquier persona que sea capaz de interpretar el modelo, podrá de nuevo obtener la realidad planteada por este. A mayor exactitud en la representación del modelo, mayor exactitud en la reproducción de la realidad. Este es un postulado básico de la ciencia.

El grado de exactitud depende de límites establecidos por razones económicas, de tiempo, de recursos tecnológicos y humanos. Parte de un buen análisis está en establecer los límites de lo que se desea representar para lograr un modelo aceptable. Tal es el caso de las ecuaciones de Newton para el tiro parabólico y la caída libre, en las cuales para obtener una representación sencilla, y sin que ello implique falta de precisión, suele despreciarse el rozamiento del aire.

Del mismo modo un buen análisis únicamente contendrá aquellos elementos que son necesarios para la operación del sistema. Los límites no sólo definen el nivel de automatización, también definirán el costo final del producto.

Sea el caso de un editor de texto, los límites de lo que este puede lograr influirán en gran medida en la complejidad del producto final, así como el costo de desarrollo y mantenimiento.



Los límites del sistema determinan su complejidad

Una computadora es la responsable de interpretar el modelo de una situación particular. Su representación se realiza a través de un algoritmo que describe el comportamiento de la realidad que se esté representando.

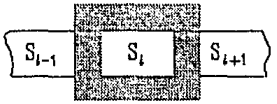
LA MAQUINA DE TURING

Alan Turing matemático inglés, propuso un modelo matemático conocido como *la máquina de Turing* con lo cual se puede resolver un problema dado en forma algoritmica.

Se puede conceptualizar a la máquina de Turing de la siguiente forma: En un cuadro de observación, siempre se tendrá un símbolo observable, en el cual se podrá leer o escribir símbolos. Este cuadro forma parte de una cinta que puede desplazarse a la izquierda o a la derecha.

La máquina observa el símbolo en el cuadro y reacciona a este con un símbolo y una acción de desplazamiento. Su reacción está determinada exclusivamente por el estado en el que se encuentre la máquina en ese momento y el símbolo observado. Estas reacciones están definidas previamente en un diccionario de símbolos y estados finito.

Se dice que es un diccionario finito porque el número de estados y el número de símbolos es finito. La máquina no puede encontrarse en el estado 1.5. Esto significa que se encuentra en el estado 1 o se encuentra en el estado 2. No puede encontrarse en puntos intermedios.



CONTROL FINITO DE LA MÁQUINA DE TURING

Una tabla de asignación de códigos de una máquina de Turing podría ser semejante a la mostrada a continuación:

ESTIMULO		REACCION		
ESTADO ACTUAL	SIMBOLO ACTUAL	ESTADO NUEVO	SIMBOLO NUEVO	MOVIMIENTO
e_{22}	s_{13}	e_{24}	s_{14}	D
e_{33}	s_{22}	e_{33}	s_{24}	D
e_{42}	s_{33}	e_{43}	s_{33}	I
e_{10}	s_{18}	e_{18}	s_{16}	I
. . .	. . .	. . .	. . .	. . .

Tabla de asignación de una máquina de Turing

En ella observamos dos partes bien definidas que conforman el diccionario. En la parte izquierda una sección dedicada al estímulo que puede presentarse ante la máquina de Turing. Dicho estímulo depende exclusivamente del estado actual y el símbolo actual. Esto dará por reacción que la máquina:

1.- Pase a un nuevo estado.
2.- Escriba un símbolo en el cuadro de observación.
3.- Efectúe un movimiento a la derecha o a la izquierda.

Es conveniente hacer notar que:
- El estado y el símbolo pueden cambiar como sucede en el primer renglón.
- El estado no cambia, pero sí el símbolo.
- El estado cambia pero no el símbolo.
- Tanto el estado como el símbolo permanecen estables.

El primer renglón se leería de la siguiente manera: *Cuando se observe el símbolo trece y la máquina se encuentre en el estado veintiocho entonces se debe pasar al estado veinticuatro, escribir el símbolo catorce en el cuadro de observación y efectuar un movimiento a la derecha.*

La máquina así descrita posee una facultad poco usual en un ingenio artificial: Puede tomar decisiones en base a los estímulos que estén presentes considerando el estado en el que se encuentre. Eso la coloca en la posibilidad de emular a un ser vivo o de servir para el modelado de un sistema de toma de decisiones. Todo lo que debe hacerse es establecer de manera correcta un diccionario que defina una acción prevista.

Colocando en la cinta la codificación correcta y luego sometiéndolo al control finito de la máquina de Turing es posible simular un proceso y computar los resultados.

Veamos por ejemplo la manera como se podría codificar la suma de dos dígitos binarios. Suponiendo que se desea que el resultado se desee en función de un dígito suma S y un dígito de acarreo C. La tabla de sumar para números binarios naturales quedaría así:

A	B	SC
0	0	00
0	1	01
1	0	01
1	1	10

Tabla de suma binaria natural

En la cinta se grabarían los dígitos A y B en forma contigua y el resultado C S quedaría exactamente en esos cuadros.

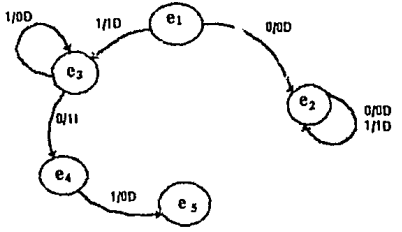
El resultado quedaría definido de la siguiente manera:

ESTIMULO		REACCION		
ESTADO ACTUAL	SIMBOLO ACTUAL	ESTADO NUEVO	SIMBOLO NUEVO	MOVIMIENTO
e_1	0	e_2	0	D
e_1	1	e_3	1	D
e_2	0	e_2	0	D
e_2	1	e_2	1	D
e_3	0	e_4	1	I
e_3	1	e_3	0	D
e_4	1	e_5	0	D

De la tabla de la suma se puede observar que si el primer dígito **A** es cero, el resultado quedará definido exclusivamente por el segundo dígito **B** y la suma será exactamente ese dígito. Esto es lo que sucede en e_1 al tener como símbolo presente un cero. Pasa al estado e_2 en donde se vuelve a escribir el dígito **B** para dar por finalizado el proceso.

En el caso de que el estado e_1 detecte al símbolo **1** entonces es necesario averiguar el valor del dígito **B.** Si este es un **1** todo lo que se requerirá será colocar un cero para dar como resultado un **1** en **C** y un **0** en **S.** Si **B** es por el contrario un **0** el resultado final debe ser cero en **C** y uno en **S** por lo que es necesario regresarse para colocar un cero en lugar de el uno que dejó el estado e_1 al pasar al estado e_3.

El gráfico muestra a cada estado encerrado en un circulo. La flecha va acompañada de una indicación con la estructura *símbolo actual/símbolo futuro Movimiento.*



Ahora veamos como funciona la máquina de Turing. En la siguiente secuencia de estados se muestra el estado actual, la cinta actual y la cinta futura. Debajo del símbolo actual y el futuro se coloca un asterisco para representar al control finito de la máquina de Turing.

	ESTADO ACTUAL	CINTA ACTUAL	CINTA MODIFICADA
1	e_1	0 1 *	0 1 *
2	e_2	0 1 *	0 1 *
3	e_2	0 1	FIN

Realmente no ha sucedido nada espectacular. La máquina ha computado que 0+1 = 01 tal y como se había esperado.

Observemos ahora la siguiente tabla:

	ESTADO ACTUAL	CINTA ACTUAL	CINTA MODIFIDA
1	e_1	1 0 *	1 0 *
2	e_3	1 0 *	1 1 *
3	e_4	1 1 *	0 1 *
4	e_5	0 1 *	FIN

En esta ocasión la máquina ha funcionado de una forma aún más interesante. Modifica los símbolos de cada cuadro y arroja un resultado correcto 1+0= 01.

Analicemos ahora los resultados obtenidos: tenemos una máquina que es capaz de realizar una acción predeterminada en base a las definiciones contenidas en su diccionario, es decir, una máquina que es capaz de tomar decisiones en base a un algoritmo planteado de antemano. Es una máquina que sirve para modelar.

¿Podríamos lograr una abstracción superior? Sabemos que la máquina que acabamos de estudiar realiza la suma de dos números. No sabe hacer otra cosa porque es una máquina de propósito especial. Pero si observamos atentamente el diccionario podemos relacionar cada estado-símbolo presente con un metasímbolo, es decir, un símbolo que representa a un símbolo. Si podemos diseñar otra máquina más compleja (MC) capaz de procesar esos metasímbolos entonces esta nueva máquina MC podría leer en una cinta los símbolos de la máquina original (MO) y simularla. Es decir MC actuaría como MO y al ser MO un modelo, tendríamos que MC sería un modelo de modelos.

Se ha logrado entonces obtener un máquina universal, ya que basta sólo con MC para simular el comportamiento de cualquier otra. Una computadora es un modelo de modelos. Basta con obtener un modelo de una situación particular, y la computadora podrá modelar a este modelo y por tanto realizar una simulación de la realidad que se quiere representar.

Regresemos un momento al gráfico de nuestra máquina MO. En ella podemos observar que existen dos estados terminales e_2 y e_5, es decir, puntos en los cuales ha decidido que el problema está resuelto. Para la máquina de Turing sólo hay dos posibilidades: El problema llega a un punto de fin o no llega. Si el problema alcanza un estado final se dice que el problema es computable o de solución algoritmica. Si no llega a un estado final entonces tendremos un *problema indecible*: sin solución algoritmica.

Alan Turing demostró en 1936 (mucho antes de que existieran las primeras computadoras verdaderas) que existen problemas indecibles, es decir, sin una solución algoritmica. Demostrar que existen problemas indecibles reviste una importancia enorme en cuanto a la computabilidad de la realidad por parte de una máquina de Turing.

En un artículo titulado *On computable numbers, with an aplication to the Entscheidungs problem* (Sobre números computables, con una aplicación a los problemas indecibles), donde el último término se refiere al problema de la decibilidad, se planteó el siguiente razonamiento:

Supóngase que se tiene una máquina particular (MP) que puede llegar a un estado final o puede no hacerlo. Construimos una máquina general (MG) que determina si MP se detiene, es decir, terminará en un estado asociado a un SI cuando MP se detenga. A su vez MG terminará en un estado asociado a un NO cuando MP no se detenga.

Por supuesto, dado que MG puede determinar si cualquier máquina MP se detiene, entonces también es capaz de determinar si ella misma MG se detiene o no.

En base a esta máquina MG construimos una máquina de ciclo MC que se detendrá cuando una máquina MP no lo haga o MC entrará en un ciclo infinito cuando MP se detenga. El ciclo infinito de MC se logrará colocando dos estados que se pasen el control en forma alternada cuando MP se detenga.

De la definición de MC podemos ver que actúa sobre cualquier máquina particular negándola. ¿ Qué sucede cuando MC actúa sobre su propia codificación? Pasará que MC no se podrá detener porque para eso es necesario que MC no se detenga lo cual no es posible. Y si no es posible, significa que MC no puede existir, negando por tanto la existencia de MG que la generó.

Este es un problema *indecible*. Este análisis es conocido como *el problema del alto de la máquina de Turing* y demuestra la existencia de problemas indecibles.

Basta con un ejemplo para demostrar la existencia de problemas indecibles y por tanto demostrar que no todos los problemas pueden ser representados en forma algoritmica.

Aunque es indemostrable , generalmente se acepta la :

Hipótesis de Turing

a) Si existe una máquina de Turing para representar un problema, entonces este tiene solución algorítmica.
b) Si un problema tiene solución algorítmica es porque existe una máquina de Turing que la representa

Las ecuaciones de Newton, o de Einstein son modelos de la realidad. La representan y la predicen. Cualquiera que tome dichos modelos, podrá reproducir el fenómeno representado. Ahora bien, desde el momento que existen problemas decibles, existen máquinas que pueden representar a dicho modelo. Una computadora es un modelo de modelos, por tanto, al representar a la realidad y al ser el modelo un algoritmo predecible, convierte a la disciplina que crea dichos programas en una Ciencia de modelado de la realidad. La computación en sí misma no puede ser una ciencia dado que no existe un objeto específico de estudio. La computación es una rama multidisciplinaria de la ciencia que toma herramientas de diversas ciencias para poder realizar los modelos requeridos. Para poder ser realmente científica es necesario que el lenguaje en el cual se exprese un algoritmo sea preciso y predecible. No se pueden admitir ambivalencias. Y es aún más deseable que los elementos de expresión de dicho lenguaje sean tan formales como los matemáticos o los químicos.

Y desde el momento que existen problemas indecibles, es necesario reconocer (en contra del sentir popular) que *una computadora NO PUEDE REPRESENTAR todos los problemas de mundo real.*

ALGORITMOS Y ABSTRACCION

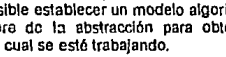
¿Cómo puede obtenerse el algoritmo de un problema dado? Esta es una pregunta obligada después de haber observado que existe una máquina capaz de modelar cualquier algoritmo que se plantee correctamente.

Básicamente, una computadora es una máquina de Turing. Cualquier cosa que una computadora pueda hacer se debe a que ha sido posible establecer un modelo algorítmico para realizar dicho resultado. El establecimiento de un modelo requiere de la abstracción para obtener sólo los puntos esenciales que realmente definen al problema sobre el cual se esté trabajando.

La abstracción trabaja con diversas herramientas que ayudan a la manipulación de los objetos sobre los cuales opera. Así en álgebra se representan a los operandos a través de letras: *x, y, z* y a los operadores aritméticos a través de símbolos que definen a las acciones: +, -, /, x.

Si lo que deseamos es trabajar con un sistema de cómputo, primero es necesario desglosar cuales son los objetos que los conforman.

Un rápido vistazo arrojará dos vistas básicas de un programa:



Y hasta hace pocos años estos eran los únicos enfoques posibles de abstraer un programa. Por un lado se tienen los datos sobre los cuales se está trabajando. La gran mayoría de los programadores y prácticamente todos los usuarios finales sólo ven los resultados de un sistema, los datos que arroja como el único punto de medición de la eficiencia de un sistema.

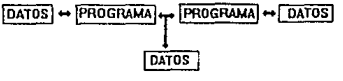
Cuando hablamos acerca de la calidad de un programa se mencionó que un buen sistema satisface las necesidades del usuario. Este requisito suele asociarse inevitablemente con los datos que el sistema da como resultado. Intuitivamente se establece que los programas deben estar hechos en función de los datos.

Este enfoque dió origen a los mecanismos de abstracción de datos como los planteados por Warnier y la metodología llamada *programación lógica*. De hecho han sido muchos los estudiosos que se han dedicado a encontrar la forma más eficiente de estructurar los datos para lograr sistemas bien definidos y eficientes en esencia.

El segundo objeto de un sistema de cómputo se visualizó hasta después: *El programa*. Bajo este enfoque un programa es un conjunto de estructuras de control que manipulan los datos. Si un programa posee estructuras de control complejas, el desarrollo y su posterior mantenimiento harán que los costos se incrementen. Un programa con estructuras de control sencillas facilitará su lectura y por ende su mantenimiento posterior.

La abstracción en el control hace referencia a las formas como se puede tener y mantener un control sencillo y eficaz dentro del algoritmo que manipula a los datos. Este enfoque eminentemente operativo hace mucho énfasis en la manera como se implantará un módulo , lo cual lo convierte en una abstracción que se debe usar hasta la fase de implantación del modelo.

Podemos inclusive tomar un nivel de abstracción que haga uso de estos dos enfoques: la *abstracción funcional:*



En dicha filosofía se observan las relaciones que establecen los programas, los controles que ejercen los módulos sobre sus subordinados y la forma como transfieren la información.

Cada una de estas formas de abstracción es en sí toda una filosofía de pensamiento. Cada uno de estos enfoques es una forma distinta de ver un sistema. Un método para abordar un problema.

Queda fuera de los límites de estos apuntes el detallar cada uno de esos puntos de vista. La rama de la ingeniería que se aboca a la obtención de los productos de programación a través de una metodología es la Ingeniería de software .

Se entiende por *producto de programación*, a los programas y documentos que apoyan, a nivel técnico o a nivel usuario, a dichos programas.

En forma idónea, las especificaciones de un programa deben estar dadas en un documento elaborado por un analista o un diseñador de sistemas. El cual puede seguir varios enfoques para abordar el problema.

La definición del problema es una labor de investigación y análisis que bien puede recordar a cualquier actividad detectivezca que se nos pueda ocurrir. La pregunta que se debe tener presente es: ¿ Qué

es lo que se desea obtener? Para responder correctamente a esta pregunta debemos hechar mano de otras dos: ¿Cuáles son las entradas del sistema? Es decir, ¿De dónde proviene la información?, ¿Cómo se genera?,¿ Cuáles son los casos de excepción?, ¿Cuál es el volúmen y la frecuencia con que se genera?, etc. La segunda pregunta es en apariencia absurda, y por lo mismo, dejada al azar con mayor frecuencia: ¿Cómo se obtiene la información de salida?

Una persona que ha comprendido realmente el problema que está tratando de modelar responde en forma concreta y sin ambiguedades a las cuestiones planteadas en el párrafo anterior. En base a las respuestas elabora un documento que recibe el nombre de *análisis de requerimientos.*

La gran mayoría de los problemas con los sistemas provienen de una mala comprensión de lo que el usuario espera del sistema de cómputo por parte de la persona que realiza el análisis de requerimientos.

Cuando se comienza a diseñar el algoritmo con el cual se resolverá un problema determinado, muchos analistas y otros tantos programadores caemos en la terrible tentación de querer tener todos los detalles de la implantación aún antes de escribir la primer linea de código. Por ejemplo, si se desea implantar un programa graficador, más de uno se sentirá tentado a pensar en los grandes problemas que representará para la rutina de despliegue el contemplar todos los tipos de monitores disponibles en el mercado... ¿Quién dijo que la rutina de despliegue debe detectar el tipo de monitor en la cual se ejecutará el sistema?. Mejor aún , ¿Quién nos asegura que existirá una rutina de despliegue?

Encontrar un algoritmo o implantarlo requiere de la abstracción de las partes más importantes y características del problema. Los detalles deben ser omitidos hasta donde sea posible.

DIVIDE Y VENCERAS

Julio César no sabía nada de sistemas de información en la forma en que los concebimos actualmente, pero de manera puramente intuitiva sabía que la mejor manera de afrontar un problema consiste en dividirlo en sus partes esenciales.

Un postulado de la Ingeniería de software se puede plantear de la siguiente manera:

Sea L(p) la longitud de un programa cualesquiera y $l(p_1)$ y $l(p_2)$ la longitud de dos segmentos mutuamente excluyentes y complementarios del mismo programa tal que:

$$L(p_t) = l(p_1) + l(p_2) \quad y$$
$$p_t = p_1 + p_2$$

Donde:

p_t es el programa en su totalidad

p_i es un programa en si mismo que conforma a p_t

Sean además $e(p_1)$, $e(p_2)$ y $E(p_t)$ los esfuerzos necesarios para obtener $l(p_1)$, $l(p_2)$ y $L(p_t)$ respectivamente, se tendrá entonces que:

$$E(p_t) > e(p_1) + e(p_2)$$

Es decir, *el esfuerzo total de un programa es menor cuando se divide en las partes que lo conforman.* De hecho se puede afirmar que:

$$E(p_t) > e(p_1)+e(p_2) > 1/2\, e(p_1)+1/2\, e(p_1)+e(p_2)$$

Lo cual es de manera intuitiva, algo correcto. Esto significa que, *para la realización de un programa o de un sistema de grandes dimensiones, la fragmentación en módulos es parte vital de un proceso de reducción de esfuerzos y por ende de costos.*

I.1.2 Identificación de los módulos (subproblemas)

Hasta este momento tenemos dos premisas sobre las cuales se puede comenzar a resolver el problema:

1.- La resolución de un programa debe hacer uso de algún método de abstracción.
2.- La partición del sistema en módulos redundará en un menor esfuerzo y un menor costo final del producto.

Analicemos el primer punto.

UNA HERRAMIENTA: EL PSEUDOCÓDIGO

El proceso de elaboración de un sistema de información de dimensiones normales (grande) es una actividad tan compleja que el cerebro no puede retener toda la información necesaria en un instante dado. El psicólogo-matemático George Miller concluyó que el cerebro humano es capaz de tratar con 7 objetos simultaneamente. Es decir 7 ideas distintas . Si se intenta pensar en más cosas a la vez se entra en un estado de confusión y olvido. Lo *complejo* de un problema sobreviene cuando se rebasa dicho límite. Con el efecto de ayudar al cerebro a plasmar las ideas, se ha hecho uso de una serie de herramientas que visualizan los cambios que una idea (un algoritmo) va experimentando en el transcurso del desarrollo de un programa.

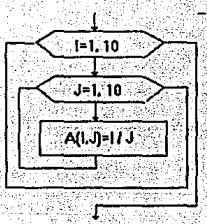
En los inicios de la computación, la herramienta más utilizada por los programadores fué el diagrama de flujo. Este es una representación gráfica de lo que la máquina realiza paso a paso y ofrece de un vistazo una panorámica completa del flujo de instrucciones que se ejecutan en ella.

Sin embargo, continuar utilizando este método para el desarrollo de programas es equivalente a seguir utilizando piedrecillas para contar, o pictogramas para expresar las ideas en forma escrita; esta última semejanza es mucho más ilustrativa, ya que el hombre en el inicio de la escritura utilizaba ideogramas y pictogramas para después llegar a una escritura más abstracta.

El diagrama de flujo tiene una serie de desventajas que han puesto en movimiento a los especialistas en la búsqueda de opciones alternas. En primer lugar, un diagrama de flujo ocupa mucho espacio en la hoja en la cual se requiera dibujar, hasta el proceso más sencillo requiere de una flecha de entrada, una caja de proceso y una flecha de salida.

Por otra parte, la simbología no siempre es uniforme y podemos encontrar muy distintos símbolos para un sólo significante. Cada vez que leemos un diagrama de flujo de otra persona o de otra compañía, es necesario averiguar primero las convenciones usadas en la representación del diagrama de flujo.

La extensión de los diagramas de flujo, su gran nivel de detalle, la gran dificultad para modificar, agregar o suprimir información los hacen muy poco prácticos. En un programa de más de 100 líneas no es posible ver de un vistazo la operación del mismo. Además los diagramas de flujo son muy dependientes del lenguaje y esto los puede volver tan obscuros como el código mismo.



El diagrama mostrado proviene del siguiente segmento de código FORTRAN. ¿ Qué es lo que ejecuta el segmento?

```
      DO 100 I=1,10
         DO 150 J=1,I
            A(I,J) = I / J
150      CONTINUE
100   CONTINUE
```

Realmente pocos programadores verán al primer vistazo que dicho segmento genera una matriz en donde todos los elementos son cero (aunque sólo se asegure dicha condición en la matriz triangular superior) y con los elementos de la diagonal principal igualados a uno. Es decir, genera una matriz identidad.

¿El diagrama de flujo aclaró en algo la función del segmento?. Definitivamente no. Por las características propias del diagrama de flujo es necesario ocupar un lenguaje taquigráfico (y generalmente una simple transcripción del código) en los símbolos ocupados en él.

La desventaja mayor, sin embargo, no es ésta. Un diagrama de flujo es muy difícil de modificar. El alto nivel de detalle que posee los convierte en un espejo del código. Pero el código está constantemente sometido a revisión y correcciones. Si el diagrama de flujo forma parte de la documentación es muy posible que en muy poco tiempo queden desactualizados. Ahora bien, si se le usa en la fase de desarrollo, las constantes modificaciones o inclusive los cambios radicales son muy difíciles de realizar. Se requiere generalmente de redibujar todo el diagrama en una hoja nueva, con la consiguiente pérdida de tiempo.

En su lugar el *pseudocódigo* se plantea como una herramienta más amable para el desarrollo y el mantenimiento de programas. Consiste de un conjunto de órdenes. No es ocioso recordar que una orden es un enunciado imperativo y consta de un : *¿Qué se hará?* y un *¿Sobre qué se actúa?*

El conjunto de órdenes del pseudocódigo están expresadas por frases cortas en español en cada una de las cuales existe uno y sólo un verbo. Una conjunción o una unión [y / o] en la frase son indicativos de que es posible desglosarlas en dos partes. La indefinición del objeto o la ambiguedad en su definición, generalmente son producto de una confusión en cuanto a lo que se quiere hacer realmente.

Debido a que las frases están dadas en español, no se tiene relación con ningún lenguaje. Se es independiente de la implantación, lo cual nos evita pensar en los detalles hasta el momento que sea necesario.

Cada segmento de código lleva un nombre, el cual subrayaremos en el presente texto. Esto nos será de gran utilidad cuando estemos definiendo los módulos.

En general es recomendable que las órdenes estén expresadas en mayúsculas y las posibles variables en minúsculas.

Las ordenes de más bajo nivel, más comunes son:

```
-LEE (variabales)
-ESCRIBE (variables)
-HAZ MIENTRAS (condicion verdadera)
 FIN DEL HAZ
-SI (condicion) ENTONCES
 SINO
 FIN DEL SI
-LLAMA proceso
```

A modo de ejemplo veamos en pseudocódigo el método tradicional para realizar un programa:

```
REALIZA PROGRAMA

ANALIZAR problema
HAZ MIENTRAS (el problema no sea resuelto)
    HAZ MIENTRAS( el algoritmo sea incorrecto)
       DISEÑAR algoritmo
       PRUEBA DE algoritmo en escritorio
    FIN DEL HAZ
    HAZ MIENTRAS (existan errores de compilacion)
       CODIFICA programa_fuente EN BASE A algoritmo
       COMPILA programa_fuente
    FIN DEL HAZ
    HAZ MIENTRAS (existan errores en tiempo de ejecución)
       EJECUTA programa
       CORRIGE errores de ejecución
    FIN DEL HAZ
FIN DEL HAZ
```

MODULARIDAD

El segundo punto con el cual se inició el epígrafe trata sobre la partición de un sistema en sus partes constitutivas. Hasta el momento hemos llegado a concluir que debido a lo complejo que un problema puede resultar para un ser humano, la partición en *módulos* es la forma más eficiente de lograr sistemas eficientes y de bajo costo.

Aún queda en el aire la pregunta: ¿Dónde y cómo se puede dividir el problema?, ¿Cuáles aspectos del problema pertenecen a la misma parte del sistema y cuáles aspectos pertenecen a diferentes partes?

Un enfoque para la resolución de estas cuestiones está en el siguiente :

PRINCIPIO DEL DISEÑO MODULAR

A.- Las partes altamente relacionadas del problema deben pertenecer a la misma pieza del sistema.
B.- Las partes no relacionadas del problema deben residir con piezas no relacionadas del sistema.

Estos es, lo que tiene relación debe relacionarse y, aquello que no presente afinidad alguna, apartarse.

Desgraciadamente las afirmaciones axiológicas (e inclusive tautológicas) de las que se valen los métodos de desarrollo para la creación de buenos programas, no son seguidas por los implantadores. Afirmaciones que a todas luces se ven obvias, lógicas, son ignoradas por los programadores que se supone, son los profesionales en la *lógica de los programas.*

El principio del diseño modular estructurado brinda la pauta para la construcción de los módulos, pero no ofrece una definición de lo que estos podrían ser. La mayor parte de los problemas de diseño modular provienen de esta confusión y malinterpretación del concepto.

Existe una anécdota que se remonta a los años sesentas. En cierta compania el jefe de departamento encomendó a sus subalternos la partición en módulos del sistema de varias decenas de miles de líneas. Una de las personas en el equipo preguntó ingenuamente sobre la medida idónea para cada módulo. Su jefe inmediato contestó que 72 líneas. Así que regla en mano, el programador midió 72 líneas y después trazó líneas rojas que separaban cada "*módulo*" de 72 líneas...

La definición de módulo varía de persona a persona y de institución a institución. La gran mayoría de estas definiciones poseen una característica común: son cuantitativas... y en principio: incorrectas.

Algunas de estas definiciones son:
- Un módulo consta del código que pueda contener una hoja de papel impreso.
- Un módulo es el resultado del trabajo que un programador es capaz de realizar en un periodo determinado (dia/semana/mes)
- Un módulo es aquel que se ejecuta en 0.5 milisegundos de procesamiento.

Algunas otras definiciones caen aún en un plano más subjetivo:
- Es una subrutina FORTRAN
- Es un procedimiento PASCAL

Lograr una definición un poco más correcta de lo que debe ser un módulo requiere de un pequeño ejercicio mental. Este consiste en un breve repaso de lo que es una caja negra.

EL CONCEPTO DE CAJA NEGRA

Una verdadera *caja negra* es un sistema que puede ser utilizada en su totalidad sin tener conocimiento de lo que hay en su interior.

La vida cotidiana nos ofrece un sin fin de cajas negras: radio, televisión, juegos de video, automóviles, etc., No es necesario conocer los detalles internos que constituyen a cada una de esas cajas negras, sino únicamente cuales son las entradas, que es necesario proporcionar para obtener una salida deseada. El estilo de vida que llevamos no sería posible si cada ser humano en la Tierra tuviera que conocer a detalle cada aparato de que dispone para su uso diario.

Así visualizada una caja negra, ésta no debe mostrar detalles de funcionameinto a quien la utilice. Esto es más propio de una caja blanca en la cual es posible ver esos detalles que resultan mucho más útiles a una persona que realiza mantenimiento.

La definición de un módulo debe aproximarse más al concepto de caja negra que a una definición meramente cuantitativa. Este es un enfoque mucho más funcional y que relaciona al principio del diseño modular al definir que un

Módulo:

Es un segmento de un programa que posee un conjunto de entradas finito, un conjunto finito de salidas y que ejecuta una sola función.

PSEUDOCÓDIGO, MODULOS Y DISEÑO PRELIMINAR

Uno de los enfoques de diseño modular más populares es el diseño arriba-abajo (Top-down), Su concepto fundamental es el siguiente:

Para iniciar un diseño modular arriba-abajo es necesario partir de las funciones mayores hacia las funciones menores. El diseño de las funciones mayores se debe realizar para el nivel más alto de abstracción (nivel 1), A su vez cada una de estas funciones se descompone en sus partes constitutivas más elementales (nivel 2). Cada una de las funciones de nivel 2 debe descomponerse en sus partes constitutivas (nivel 3). Esta partición continúa hasta el momento en que se tengan funciones fácilmente codificables.

El nivel de detalle necesario queda entonces determinado por la *facilidad* con la cual es posible codificar el módulo. Por supuesto que esta facilidad está en función de la experiencia del programador o de sus conocimientos del lenguaje en el cual desea implantar.

Pongamos a título de ejemplo que se desea implantar una función cualquiera que llamaremos **TRIZ**. Tan pronto como tengamos correctamente definidas las entradas, el proceso y las salidas deseadas, estaremos en condiciones de comenzar a desglosar la función **TRIZ** en sus componentes esenciales.

```
TRIZ
ACEPTA datos
EDITA datos
EFECTUA TRANSACCION
IMPRIME RESULTADOS
```

Esas son las funciones más generales que constituirán al sistema **TRIZ**. podemos tomar una de ellas y comenzar a particionarla:

```
ACEPTA_DATOS
INICIALIZA variables
ABRE archivo_maestro
BUSCA fin_de_archivo
HAZ MIENTRAS ( se desee continuar capturando)
    DESPLIEGA pantalla_captura
    PIDE campo_llave
    AÑADE campo_llave
    PIDE otros_datos
FIN DEL HAZ
CIERRA archivo_maestro
RETORNA
```

Debe observarse que aún no se ha pensado o especificado ningún lenguaje, algunas de las sentencias son demasiado generales y podrían o no ser directamente ejecutables. Sin embargo, los detalles aún no deben preocuparnos. Aún estamos en un nivel de abstracción tal que nos permite omitirlos.

Son muchos los programadores que se enfrascan en los detalles y vuelven una tormenta de un aspecto trivial del sistema. Esto hace que se deje de ver el bosque por ver los árboles.

I.1.3 REFINAMIENTO SUCESIVO DE LOS MÓDULOS

Tal y como fué expresado por Wirth este concepto puede verse de la siguiente manera:

> *En cada paso del refinamiento, una o varias instrucciones del programa dado se descomponen en una o más instrucciones detalladas. Esta descomposición sucesiva o refinamiento de especificaciones termina cuando todas las instrucciones están espresadas en términos del computador usando un lenguaje de programación... Conforme se refinan las tareas, también los datos pueden ser refinados, descompuestos o estructurados, y es natural refinar el programa y las especificaciones de los datos en paralelo.*

> *Cada paso de refinamiento implica algunas decisiones de diseño. Es importante que... el programador sea consciente de los criterios subyacentes (en las decisiones de diseño) y de la existencia de soluciones alternativas.*

Un sistema definido de esta manera puede representar en cada sentencia individual a una función. Sin embargo, un aspecto al que generalmente no se le presta la importancia debida es la que se refiere a los parámetros entre módulos padre y cada uno de sus hijos.

Retomando la ecuación del esfuerzo vista anteriormente se tenía:

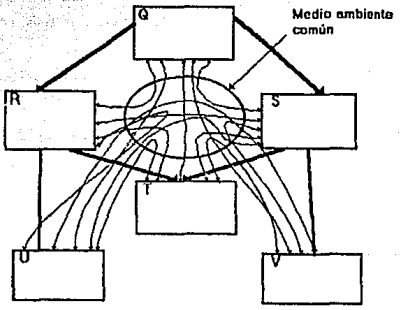
$$E(p_i) > e(p_1) + e(p_2) > 1/2\, e(p_1) + 1/2\, e(p_1) + e(p_2)$$

Esta ecuación sin embargo, suponía que el segmento p_1 era complementario y excluyente del segmento p_2. Es decir, no existe comunicación de datos, ni existe el traslape de funciones o de código. Un programa que debe comunicarse con otro, lleva a la necesidad de crear código que funcione como interface entre los módulos. Y por supuesto requiere de un esfuerzo adicional para definir cuales serán dichas interfaces. De donde se obtiene lo siguiente:

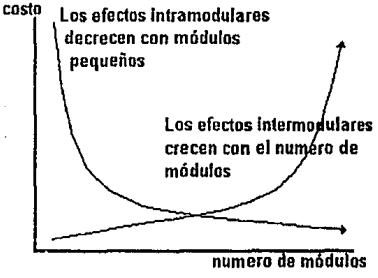
$$E(p_i) > e(p_1) + e(p_2) + e(I) > e(p_1) + e(p_2)$$

Donde

e(I) es el esfuerzo invertido en la interface



Cuando las variables comparten un medio común, las relaciones se toman muy complejas



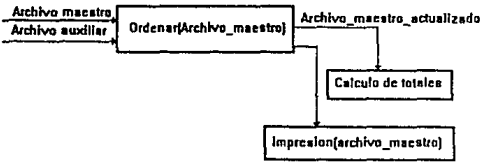
Influencia sobre el costo de las relaciones intermodulares e intramodulares

ACOPLAMIENTO

Una de las metas fundamentales del diseño modular es la flexibilidad que adquiere cada uno de los elementos constitutivos. Esta flexibilidad se expresa en función de la facilidad que presentan al cambio y que tanto afectan dichos cambios a otras partes del sistema.

La definición que hemos expresado sobre un módulo indica una característica que debiera ser propia de un buen diseño modular: la *independencia.*

Supongamos un subsistema de actualización de archivo maestro. En dicho subsistema, después de la apertura , mezclas y borrados que fueran pertinentes para la aplicación específica, encontramos un módulo de reordenamiento de los datos.



En dicho archivo se mantienen en forma física registros dados de baja en forma lógica por los operadores. Es decir, aunque un registro ya no se vea, ni sea tomado en cuenta por el sistema para efectos de cálculo, aún permenece ocupando espacio dentro del archivo maestro. Una vez que se ha realizado la apertura del archivo maestro se eliminan los registros en forma física.

El módulo encargado de reordenar los datos pide como entrada un archivo maestro, un archivo auxiliar de características predefinidas y como salida un `archivo_maestro_actualizado`. El `archivo_auxiliar` es creado por el módulo padre del ordenador y debe encontrarse vacio...

¡Bueno así funciona este módulo ordenador!. Nosotros, desde este punto, no deberíamos de preocupamos por los detalles de implantación, sin embargo, y para sorpresa del diseñador, el módulo que hemos citado tiene, a este nivel, detalles de implantación que pueden resultar muy caros.

Suponer que se requiere de un archivo auxiliar con las características mencionadas, es restringir el método de ordenamiento a usar ese archivo, o a ocupar un método que haga uso de él. Y más peligroso aún, exponer al módulo a que cualquier módulo padre use el archivo auxiliar y lo deje en un estado inconveniente para su uso por el ordenador. Después de todo ¿Quién va a suponer que un archivo, vacio y sin indicios de ser necesitado pueda afectar a otra parte del sistema?.

Una modificación tal, que nos permitiera escribir un método más eficiente en cuanto a velocidad y recursos de memoria utilizados podría prescindir de dicho archivo, lo cual puede generar o no problemas, dependiendo del tratamiento posterior de dicho archivo. Podría existir un módulo que usara el producto intermedio generado por el archivo auxiliar.

Dentro del diseño modular se debe cuidar que cada módulo sea una pieza intercambiable. Tal es la filosofía dentro del hardware en donde basta cambiar un módulo(una tarjeta) para convertir la computadora en otra con más características, más poderosa y aprovechando los recursos existentes.

Esto es posible lograrlo en mayor o menor escala, pero depende de que tan ligado esté un módulo con el resto del sistema. Si es un menor número de parámetros los que componen al módulo, entonces será más sencillo modificarlo o inclusive reemplazarlo.

Un caso típico se presenta en el siguiente ejemplo. En la compania XYZ se desea modificar al módulo A, así que se le asigna el trabajo a un programador. Muy pronto el programador descubre que para entender al módulo A debe comprender como se procesa tal cosa en el módulo B. Y que para comprender al módulo B también debe entender al módulo C, creando así una larga y frustrante cadena que lo único que logra es retrazar sensiblemente la entrega a tiempo del problema.

De lo dicho anteriormente podemos observar que de la forma como se comuniquen los datos queda determinada la flexibilidad de un sistema. La claridad con que se definan tanto los parámetros como la función determinará la reutilización de código para aplicaciones futuras.

Se ha insistido mucho en la necesidad de una correcta definición de los parámetros y de las funciones que un módulo debe desempeñar. Los errores más costosos dentro de cualquier proyecto de programación son los de diseño. Un buen diseño redundará en menores costos y más facilidades en el resto de las labores de implantación y mantenimiento.

En el ejemplo citado anteriormente se habló del conocimiento que el programador necesitaba de B o C para poder modificar A. Sin embargo el *conocimiento* no es un término fácilmente cuantificable. Sabemos que A está acoplado a C pero está más altamente acoplado a B. Esto hará que el programador ocupe de mayor tiempo al tener que conocer también B para modificar A.

Con el término *acoplamiento* se hace referencia al grado de interdependencia entre módulos. Y según se sigue de la discusión anterior el acoplamiento entre módulos puede ser considerado dentro de una escala del más fuerte (el menos deseable) al más débil (el más deseable) de la siguiente forma:

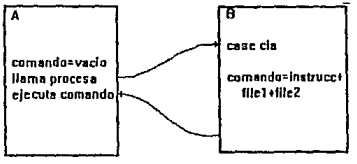
- Acoplamiento por contenido
- Acoplamiento por zonas compartidas
- Acoplamiento de control
- Acoplamiento de zonas de datos
- Acoplameinto de datos

Acoplamiento por contenido

Un módulo modifica los valores locales o las instruciones de algún otro.

Básicamente este tipo de acoplamiento se presenta en programas hechos en lenguaje ensamblador, en donde es posible que se escriban datos sobre cualquier área de memoria.

Sin embargo existe una forma más sutil de acoplamiento por contenido que es utilizada por muchos programadores. El uso de *macroinstruciones*. El uso de variables que contienen instrucciones puede llegar a extremos de verdadero abuso. Imaginemos un módulo padre A que hace una llamada a cualquier módulo B de lectura o procesamiento.



En el cual a través de variables de ambiente (por contexto) se establece la acción más pertinente a ser ejecutada. Esta acción se almacena en la variable **comando**, la cual es devuelta al módulo padre para que sea ejecutada.

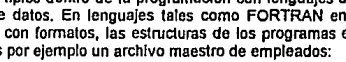
Esta es una relación claramente patológica. El módulo A es jerárquicamente superior a B, el módulo A es padre del módulo B, sin embargo el comportamiento de A está supeditado a B. El módulo B le dice a su padre que es lo que debe hacer.

Esta relación de acoplamiento es muy fuerte. No será posible modificar alguno de los módulos en forma separada porque ambos están estrechamente unidos.

Acoplamiento por zonas compartidas

Los módulos están atados en forma conjunta por medio de sus zonas globales para las estructuras de datos.

Este es un caso muy típico dentro de la programación con lenguajes de alto nivel que no hacen uso de manejadores de bases de datos. En lenguajes tales como FORTRAN en los cuales las lecturas a los archivos deben especificarse con formatos, las estructuras de los programas están ligadas a las estructuras físicas del archivo. Pongamos por ejemplo un archivo maestro de empleados:



Cualquier acceso al archivo maestro supondrá los formatos especificados. De tal forma que una rutina que desee accesar el Departamento al cual pertenece el empleado deberá especificar un espaciado de 5+10+15+15 = 45 caracteres antes de leer el dato. Cualquier cambio a esta estructura supondrá también el cambio de todos los formatos de lectura y escritura.

Y esto que se hace evidente para el manejo de los archivos es ignorado por los programadores al elaborar sus programas haciendo uso de variables globales.

En muchas ocasiones los programadores suelen definir variables que serán de uso extensivo en todo el sistema. Por ejemplo, definir un arreglo con los nombre de los doce meses del año. Dado que cualquier módulo puede hacer uso de dicho nombre es fácil declararlo una vez y usarlo siempre... Imaginemos un arreglo definido de la siguiente manera:

```
ANNO="1992"
MES[1]="ENE"
...
MES[12]="DIC"
...
```

De tal forma que la expresión siguiente :

```
FECHA=MES[I]+" / "+ANNO
```

produjera:

FECHA=> ENE / 1992
FECHA=> DIC / 1992

La longitud normalizada a tres caracteres del nombre de cada mes no es problema para la correcta concatenación del mes y el año y, siempre produce una fecha de una longitud determinada.

A la manera de los cuentos podemos decir: un día llegó a la compañía un programador nuevo, amante de las ventanas de opción y diálogo, al cual le encargaron escribir un programa que realizara la captura de ciertos parámetros del sistema, lo cual incluía el mes de trabajo. Nuestro programador, ante las insistentes quejas de los usuarios respecto a los mensajes crípticos y taquigráficos usuales del sistema, decide colocar en el menú de lectura de los meses, el nombre completo de cada uno. Y para ello lo único que realiza es redefinir el arreglo de los meses. De esta manera su menú de opciones sufre la transformación mostrada a continuación:

ENE	ENERO
FEB	FEBRERO
MAR	MARZO
ABR	ABRIL
MAY	MAYO
JUN	JUNIO
JUL	JULIO
AGO	AGOSTO
SEP	SEPTIEMBRE
OCT	OCTUBRE
NOV	NOVIEMBRE
DIC	DICIEMBRE

Los usuarios quedan muy complacidos con el cambio. Sin embargo, los efectos secundarios no se harán esperar. Muchos de los reportes llevan la fecha de emisión justificada a la derecha . El cambio introducido termina por producir un error en todos esos reportes:

PAG 1 DIC / 1992	===>	PAG 1 DICIEMBRE
PAG 1 MAY / 1992	===>	PAG 1 MAYO / 19

El uso de este tipo de variables puede causar muy serios problemas. Su uso debe estar clara y plenamente justificado. Programas con variables de tipo global limitan muy seriamente su independencia ya que requieren de datos definidos extralímites de la función. En el capítulo dos se pondrá un ejemplo sobre su uso y las características que deben cubrir para ser lo menos nocivas para el sistema.

Acoplamiento de control

Un módulo controla la secuencia de otro a través de banderas de control.

Observemos el siguiente segmento de pseudocódigo correspondiente a un programa de juego:

```
P.PRINCIPAL DEL JUEGO
HAZ MIENTRAS ( continuar)
    SI ( juega_maquina ) ENTONCES
         TIRA_COMPUTADORA(juega_maquina)
    SINO
        ACEPTA_JUGADA(juega_maquina)
     FIN DEL SI
FIN DEL HAZ
```

En donde:

```
TIRA MAQUINA
CALCULA JUGADA MAQUINA
DESPLIEGA JUGADA
NIEGA QUE juega_maquina
```

```
ACEPTA_JUGADA
HAZ MIENTRAS ( no sea jugada valida)
    PIDE jugada_usuario
FIN DEL HAZ
DESPLIEGA JUGADA
AFIRMA QUE juega_maquina
```

Dentro del ciclo de repetición se encuentran dos llamadas a funciones excluyentes. Una calcula una jugada de algún juego por parte de la computadora y la otra acepta una jugada válida del humano.

Para que se pueda conmutar entre un módulo u otro se hace uso de la bandera `juega_maquina` de tal forma que si está afirmada se ejecuta la rutina `TIRA_COMPUTADORA`, pero si está negada se ejecutará `ACEPTA_JUGADA`.

Aquí vemos con bastante claridad que el ciclo no operaría correctamente si la bandera no se actualizara en los módulos hijos. De hecho su operación es totalmente dependiente de lo que suceda con la bandera.

En un programa hijo con 4238 líneas se podría perder de vista que la línea 3043 que actualiza la bandera es indispensable y borrarla como parte de un segmento que ya no es útil.

Acoplamiento por zonas de datos

Es similar al de zonas compartidas, pero los datos son compartidos en forma selectiva por las rutinas.

Esta no es una forma común de acoplamiento. Se presentaba en los programas FORTRAN en la estructura COMMON. A través de este declarador es posible crear áreas de datos comunes entre subrutinas. Cada área de datos puede tener un nombre y ser utilizada y referenciada por dicho nombre:

```
COMMON /DATOS/MESES(12), COMPANIAS(3), RFC(3)
COMMON /PANTALLAS/MENU(4),COLORES(12)
COMMON /REPORTES/CODIGOSIMP(7), TIPOSLETRA(4)

SUBROUTINE CHEQUES
COMMON /DATOS/ MESES(12), CIAS(3),REGISFC(3)
COMMON /REPORTES/CODIMPRESORA(7),FUENTES(4)
```

En este ejemplo la subrutina cheques no recibe ningún parámetro formal, pero los datos son pasados a través de áreas de datos comunes con el resto del sistema: /DATOS/ y /REPORTES/.

COMMON únicamente pàsa un apuntador al primer elemento de la lista, de tal forma que es posible redefinir nombres o hacer manipulaciones poco recomendables como:

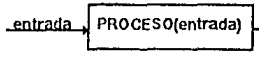
```
COMMON /DATOS/ENE,FEB,MAR,ABR,MAY,JUN,JUL,AGO,
*       SEP,OCT,NOV,DIC,CIA(3),RFC1,RFC2,RFC3
```

Como se estudiará en su momento en el capítulo dos, existe una forma similar de COMMON en lenguaje C : la union. Esta debe ser tratada con sumo cuidado (Vease capítulo II)

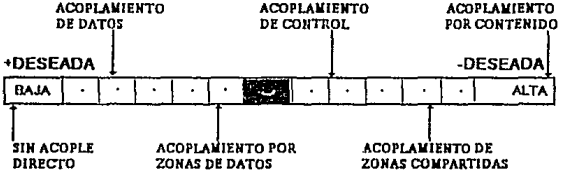
Acoplamiento de datos

Incluye el uso de listas de parámetros para pasar los elementos entre rutinas.

Esta es la forma más recomendable. Al pasar los datos por medio de listas definidas explícitamente, cualquier programador sabrá de donde proceden los datos que son procesados en un módulo cualquiera, es más, De la definición y la notación empleada hasta este momento, es la forma más formal para definir un módulo. En realidad la notación de módulos bajo expresiones puramente matemáticas es la manera más formal de representar un módulo.



Este también permite establecer que parámetros son realmente necesarios y cuales pueden ser generados en otros módulos. Por ejemplo en el ejercicio presentado anteriormente sobre la bandera de control, es relativamente sencillo suponer que dicha bandera es inecesaria, La linea que la actualiza debe ir fuera del módulo y no dentro del mismo. Se llegará a esta conclusión cuando se establezca la función del módulo y los parámetros que son requeridos.



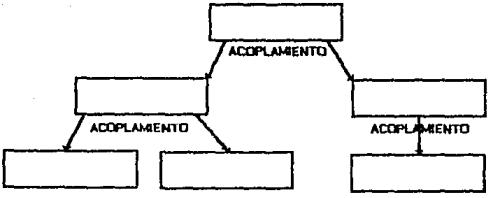
Tipos de acoplamiento

Una variación de acoplamiento está vinculado con la portabilidad del código fuente. El encadenamiento del sistema a un tipo específico de equipo o a un compilador específico, pueden crear una dependencia con gran cantidad de estragos cuando sea necesario cambiar el equipo o se adquiera una versión distinta del compilador.

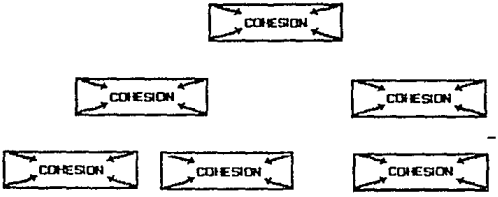
No estamos satanizando los distintos tipos de acoplamiento y restringiendo la libertad de diseño, en ocasiones es por demás indispensable valerse de algún tipo de acoplamiento que no es precisamente el recomendable, sin embargo, es importante señalar que en el momento de diseñar se deben sopesar las implicaciones que dicho tipo de acoplamiento acarraerá con posterioridad.

COHESION

En tanto que el acoplamiento se refiere a la fuerza de relación entre módulos de un sistema. La cohesión interna de un módulo se mide en términos de la fuerza de unión de los elementos dentro del módulo.



ACOPLAMIENTO: Fuerza de relaciones entre módulos



COHESION: Fuerza de relacion dentro de los módulos

Los conceptos de acoplamiento y cohesión están íntimamente relacionados. A mayor cohesión de cada módulo en el sistema, menor será el acoplamiento que existirá entre ellos. En muy buena parte esto es intuitivamente evidente. Durante el desarrollo de un módulo es necesario pensar siempre en el **Principio del diseño modular.** Entre más relacionados estén los componentes del sistema, menor relación tendrán con otros módulos.

La cohesión es tomada por muchos diseñadores en una forma muy asociativa y que puede hacer perder de vista el principio de diseño modular. Tal es el caso del diseñador que se dice: *Bueno, el módulo A está relacionado con B, a su vez este tiene que ver con el módulo C, así que los junto a todos y así me ahorro pasar parámetros. ¡Genial!.* Sin embargo el resultado final nos lleva a una cohesión del módulo resultante (en su totalidad) muy baja debido a que A no tiene nada que ver con el módulo C.

Existen siete niveles de cohesión y que listados dentro de una escala de la más débil (la menos deseada) a la más fuerte (la más deseada) son:

- Cohesión coincidental
- Cohesión lógica
- Cohesión temporal
- Cohesión de comunicación
- Cohesión secuencial
- Cohesión funcional
- Cohesión informacional

Cohesión coincidental

Los elementos que forman el módulo no tienen relación aparente entre cada uno de ellos.

Han aparecido ya dos ejemplos de este tipo de cohesión a lo largo de estas notas. El primero fué la anécdota en la cual se *modularizaba* simplemente dividiendo un código monolítico en segmentos de igual tamaño. Es por demás evidente que los elementos de procesamiento quedarán unidos de forma arbitraria y que no tendrán relación unos con otros.

Un segundo ejemplo más típico y sutil fué la rutina que devuelve la bandera **juega_maquina**. Un módulo coherente debe realizar una sola función, y el hecho de que la línea que pasa el control al otro jugador sea eliminada del módulo no afecta la función que ejecuta ese módulo (generar una jugada válida) . Un programador que al revisar un módulo encuentre asignaciones que no son utilizadas en ninguna parte de *ESE* módulo siempre se sentirá tentado a borrar dichas líneas, que bien pueden ser consideradas basura.

Sin embargo existe un caso aún más típico y problemático. Un programador ha visto que un segmento de código cualquiera se utiliza en varias partes de su programa.

Supongamos por ejemplo que:

```
SI ( bandera4 es cierta) ENTONCES
    SALTA AL SIGUIENTE REGISTRO
    INCREMENTA 1  LINEA DE IMPRESION
    REINICIALIZA suma_parcial
FIN DEL SI
```

aparece en diversas partes, así que nuestro programador decide crear un módulo llamado `MOSI` que le evite escribir estas líneas y, de paso, ahorrar un poco de código.

Sin embargo, en un caso particular no es necesario reinicializar la suma, o bien, es necesario incrementar 2 líneas de impresión. Si un programador se encuentra depurando precisamente ese caso específico le resultará muy sencillo de modificar `MOSI` para ejecutar lo que él desea, aunque se producirá un error en todas las otras rutinas que usaban a `MOSI`

Cohesión lógica

Implica algunas relaciones entre los elementos de un módulo. Los módulos lógicamente unidos normalmente requieren de una descomposición mayor.

Un ejemplo de módulo lógicamente unido normalmente incluye funciones que al relacionarse crean estructuras muy complejas e interrelacionales. En el módulo `LEETODO` podría crearse un bloque monolítico para determinar el tipo de dispositivo de entrada así como el manejo de los diversos errores.

Las banderas de error, las decisiones y los parámetros pueden crear relaciones bastante complejas. Cada uno de los segmentos que manejen un dispositivo podría abstraerse como un módulo. Si se ve bajo esta perspectiva, cada módulo realiza una función y estará fuertemente acoplado con los demás módulos.

Suelen reunirse en un solo módulo funciones que tienen alguna relación lógica. En el módulo de altas se tendrán juntos segmentos que dan de alta un archivo maestro o los archivos detalle. Aunque realizan la misma función lógica *dar de alta* es indispensable recordar que un módulo debe hacer una sola cosa sobre un objeto específico.

Dentro de la escala de cohesión, este tipo de unión se muestra más arriba debido a que los elementos están unidos por un criterio común, cosa que no sucede dentro de la cohesión coincidental.

Cohesión temporal

Todos los elementos son ejecutados en un momento dado sin requerir de ningún parámetro o lógica alguna para determinar que elemento debe ejecutarse.

Prácticamente en cualquier sistema es común encontrar por lo menos un módulo de *inicialización* : apertura de archivos, inicialización de variables, declaración de parámetros generales de trabajo, etc.

Dentro de la escala de cohesión, un módulo con este tipo de unión entre sus elementos se encuentra más arriba con respecto a la cohesión lógica porque en un módulo de cohesión temporal los elementos se encuentran unidos a través de una relación de tiempo. Es necesario que la inicialización ocurra en un periodo bien específico de tiempo, de tal modo que no es fácil suponer que alguno de los elementos se ejecute en algún otro segmento de programa. Esta asociación en el tiempo da una mayor cohesión con respecto a los otros tipos vistos anteriormente. Sin embargo en la escala completa aún posee una cohesión baja.

Este tipo de unión por otra parte puede llegar a representar importantes problemas. Es común hacer una serie de suposiciones que no necesariamente tienen por que cumplirse.

Veamos el siguiente segmento de código:

```
RESERVA AREA_1 PARA EL archivo_A
RESERVA AREA_1 PARA EL archivo_B
ASIGNA CERO A valor inicial
ASIGNA valor_inicial A contador, total, subtotal,numero
DECLARA codigos COMO ARREGLO DE 50 ELEMENTOS
HAZ DESDE INDICE= 1 HASTA 50
     ASIGNA BLANCOS A codigo(indice)
FIN DEL HAZ
```

En él se declara que se usará la misma área de memoria para los archivos A y B ... aunque nadie asegura que en algún tiempo a futuro se requiera usar ambos archivos simultáneamente. Aquí como en varios lenguajes, el pseudocódigo asigna simultáneamente un cero (valor asignado a la variable `valor_inicial`) a cuatro variables. Si por alguna razón se deseara cambiar a 1 `valor_inicial`, simplemente modificando la línea tres esto repercutirá en el ciclo que no inicializaría correctamente al arreglo `código`.

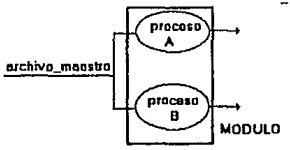
Este es un ejemplo demasiado pequeño para que pudiese ocurrir dicha eventualidad; pero en un módulo de 120 líneas, entre la asignación de `valor_inicial` y la asignación de las cuatro variables podrían existir 20 ó 30 sentencias previas, y otras 20 ó 30 entre ésta y el ciclo, de tal forma que es fácil perder de vista aspectos como el que se ha mencionado.

Cohesión de comunicación

Los elementos se refieren al mismo conjunto de datos de entrada o salida.

Segmentos de código que no necesariamente sean interdependientes pueden en un momento dado compartir los mismos datos. Por ejemplo: Un módulo que imprima en papel y envíe a una cinta la información contenida en algún archivo maestro. Aquí, aunque el proceso de impresión no depende del proceso **GRABAR_CINTA** ambos utilizan los mismos datos.

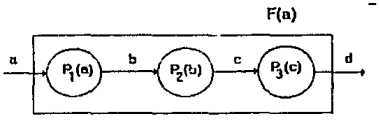
Esta asociación de módulos por medio del factor común (datos) hace más fuerte la cohesión. Se depende de uno de los pilares del sistema (los datos) aunque los segmentos que conforman el módulo no tengan mucho que ver el uno con el otro.



Cohesión secuencial

Ocurre cuando la salida de un elemento es la entrada para el siguiente.

Al igual que en la cohesión por comunicación, la cohesión secuencial hace referencia al mismo conjunto de datos. Solo que después de haber sido procesados por un segmento precedente. En términos de los datos, éstos van siendo procesados en cadena por los segmentos que conforman al módulo.



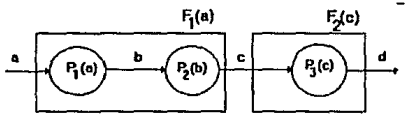
$$d=F(a)=P_3(P_2(P_1(a)))$$

Aunque en la figura se aprecia en su forma más abstracta la secuencia de los procesos que conforman al módulo, la bifurcación en más de una rama de flujo de datos es, en la mayor parte de las ocasiones, necesaria.

Módulos con este tipo de unión son producto de un enfoque claramente orientado al flujo de datos. Sin embargo, es común con un enfoque orientado al flujo de control. En donde este se refiere a las líneas que indican a donde debe pasar el control después de ejecutar determinado elemento de procesamiento.

Analizando la función F(a) definida en la figura observamos que el proceso P_1 recibe el dato a y lo entrega como el dato **b** al proceso P_2 (b). P_2(b) a su vez genera el dato **c** que es un insumo del proceso P_3(c) para dar por resultado a d.

Un diseñador típico tendrá originalmente tres procesos P_1 (), P_2() y P_3(). Estos tres pueden ser una sola función F(a), tal y como se ha mostrado. El diseñador, sin embargo, podría crear dos funciones F_1(a) y F_2(c) de tal modo que se redefiniría al sistema como:



$c=F(a)=P_2(P_1(a))$ $d=F_2(c)=P_3(c)$

Al definir dos módulos en lugar de uno solo, los conceptos de acoplamiento vuelven a tener vigor. Un mal diseño para este caso en particular podría excluir de la función F_1(a) a proceso P_2(b) y reunirlo con el proceso P_3(b) para formar F_2(a). La decisión sobre que debe ir reunido en el mismo módulo vuelve a quedar definida por el **Principio de diseño modular** y los aspectos funcionales del sistema.

Cohesión funcional

Todos los elementos se encuentran relacionados para el desempeño de una función.

Desafortunadamente la definición parece ser bastante ambigua y la siguiente, bastante subjetiva:

En un módulo completamente funcional cada elemento de procesamiento es una parte integral y escencial para la ejecución de un módulo.

Para establecer una definición más operacional podríamos establecer que:

Una función con cohesión funcional es aquella que no posee cohesión secuencial, comunicacional, temporal, lógica o coincidental.

Es un módulo que no posee elementos ajenos a la función que desempeña, la cual debe ser única y sencilla.

Ejemplos de ello vienen del enunciado con el cual se define a la función: *Lee dato del tipo XYZ de la ventana de diálogo*, especifica claramente lo que el módulo debe realizar. En cambio un enunciado como *Obtén todos los registros de entrada*, es demasiado ambiguo porque no especifica claramente que o de donde provienen los registros de entrada.

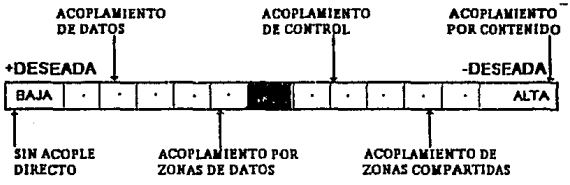
Constantine sugiere lo siguiente para establecer el nivel de cohesión de un módulo.

> *"Encontramos que una manera efectiva de hacer esto es describir total y precisamente, la función de un módulo en una sentencia sencilla en Español [...] si el módulo es en naturaleza funcional, será posible describir su operación totalmente con una sentencia imperativa de estructura simple; usualmente con un simple verbo transitivo y un objeto específico singular. Además, las siguientes pautas pueden ser usadas para guardar a distinguir los módulos no funcionales:*
>
> *1.- Si la proposición tiene que ser compuesta y que contenga una coma o más de un verbo, el módulo probablemente estará desempeñando más de una función, por lo que tal vez tenga una unión del tipo secuencial o de comunicación.*

2.- Si el enunciado contiene palabras relacionadas con el tiempo tales como ***primero, despues, entonces, cuando, y principio entonces*** *el módulo probablemente tiene una unión secuencial o temporal.*

3.- Si el predicado de la oración no contiene un objeto específico y sencillo que le siga al verbo, el módulo probablemente se encuentre unido lógicamente; por ejemplo ***Edita todos los datos*** *está unido lógicamente mientras que* ***Edita los datos fuente*** *puede tener una unión funcional.*

4.- La existencia de palabras como ***inicializa*** *y* ***limpia*** *denotan una unión temporal."*



Acoplamiento de datos

I.1.3.1 Subcódigo y diagramas estructurados

Las relaciones entre los módulos pueden representarse a través del uso de *diagramas estructurados o diagramas de estructura.* Esta es una herramienta de diseño que muestra las relaciones de comunicación y acoplamiento que el sistema va adquiriendo conforme va surgiendo en la etapa de diseño. En esta herramienta se muestran los datos que sirven como interfaces entre los diversos módulos del sistema y, por supuesto, se presentan los módulos involucrados.

Esta herramienta forma parte de una metodología conocida como *Análisis Estructurado* y no necesariamente se tiene que aplicar a los programas diseñados bajo los principios estructurados que se presentarán más adelante.

Como toda herramienta de diseño, su principal función estriba en poner los detalles de comunicación de los módulos antes de que ninguna línea de código sea escrita. Con esto no se pretende que describan el algoritmo de solución de un determinado problema, no describen una lógica procedimental de ningún tipo. Esta tarea está reservada al pseudocódigo o en su defecto a los diagramas de flujo. Tampoco determinan el tipo de interface real que se utilizará en las llamadas entre módulos. En cambio indican las relaciones de acoplamiento y permiten visualizar posibles problemas potenciales de diseño en ese aspecto.

Los símbolos empleados en los diagramas de estructura se encuentran estandarizados en la literatura que aborda el tema. Se representa a un módulo del programa como un rectángulo que lleva inscrito el nombre del módulo. Una flecha representa una llamada a otro módulo y no necesariamente están relacionadas con un mecanismo particular.

En la figura podemos apreciar un ejemplo de lo expuesto hasta este momento. Se puede ver un módulo llamado `música` que hace una llamada a un módulo hijo llamado `nota`, la comunicación se establece entre ambos módulos a través de los parámetros `frecuencia` y `duración`. Dichos parámetros determinarán la conducta del módulo hijo. Los parámetros se representan con una flecha cuya base está formada por un círculo vacío. La punta de la flecha indica la dirección en la cual viajan los datos (del módulo padre al módulo hijo).

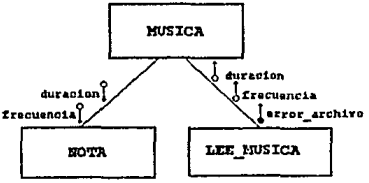
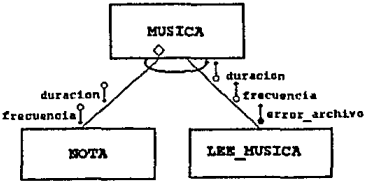


Diagrama estructurado

Cuando un módulo interactúa con su padre las flechas se encargan de establecer esta situación indicando el retorno de un parámetro como resultado de la acción efectuada.

Un módulo puede controlar a más de un módulo hijo. Cuando ésto sucede se suele indicar el tipo de sentencia de control que efectúa la llamada. Una llamada establecida a través de un punto de decisión se indica colocando un pequeño rombo en la base de la flecha de llamada. Un módulo que sea llamado en un ciclo iterativo se representa con una flecha en arco. En el ejemplo mostrado el módulo `musica` lee las notas de un archivo e inmediatamente después las emite como un sonido audible. Pero como la nota leída del archivo pudiera ser errónea en frecuencia, o bién, suceder un error en la lectura del archivo, se restinge su ejecución en una decisión sobre la validez de los parámetros que le serán pasados.

Cuando un módulo hijo retorna al programa padre un parámetro de control (una bandera) como lo podría ser un aviso del éxito o fracaso del trabajo encomendado, el parámetro se representa con una flecha que contiene en su base un círculo lleno.



Debe observarse que aunque el diagrama de estructura hace referencia a un tipo de estructura de control del programa, no es la representación de dicho algoritmo. Un diagrama de flujo es una representación del orden en que serán ejecutadas las instrucciones en el tiempo. Esto significa que un diagrama de flujo es dependiente del tiempo. Un diagrama de estructura es atemporal. Esto es porque solo representa asociaciones de control y secuencias de control.

I.1.3.2 El árbol y la tabla de decisión.

Dentro del diseño de un sistema o de un programa es muy importante validar todas las posibles ramificaciones que una situación particular puede presentar. El modelo no podrá estar completo si no contemplan todas las decisiones que el programa pueda tomar durante su ejecución cotidiana. En las novelas de ciencia ficción, las máquinas cibernéticas pierden la batalla siempre que se tienen que enfrentar a situaciones inesperadas. Dado que no han sido programadas para actuar en situaciones extraordinarias no saben tomar las decisiones adecuadas. En la vida real las máquinas no pierden ninguna batalla... es el programador el que las lleva de perder.

Para evitar este tipo de situaciones se dispone de herramientas que nos ayudan a plantear la mayor parte de las situaciones por las cuales tendrá que atravesar el sistema en funcionamiento.

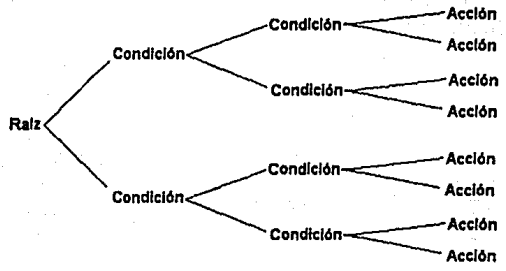
ÁRBOLES DE DECISIÓN

El *árbol de decisión* es una representación gráfica y secuencial de las condiciones a las cuales se tendrá que encarar un sistema y de las acciones que se deberán tomar para cada una de ellas.

Un árbol de decisión se escribe colocando en la parte izquierda las condiciones o los estados que adquiere el sistema y en la parte derecha se colocan las acciones que se tomarán para cada una de dichas condiciones.

CONDICIONES				**ACCIONES**
Estados posibles de los eventos	⇨	Nos llevan a	⇨	Alternativas, pasos, actividades o procedimientos que deben emprenderse cuando se toma una decisión específica

La raíz del árbol es la parte a la extrema derecha y es el punto en donde inicia la toma de decisiones que determina al algoritmo en análisis. Conforme se avanza a la derecha van apareciendo condiciones que ramifican al árbol de decisión en múltiples acciones que deben ser llevadas a cabo cuando dichas condiciones se cumplan. Cada punto de toma de decisión es conocido como *nodo* del árbol.

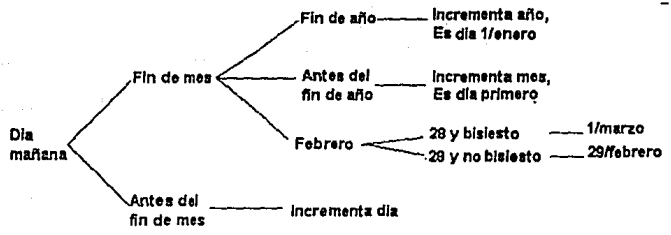


Dentro del análisis de un sistema es imperiosa la necesidad de presentar de manera formal las decisiones que el sistema necesite tomar y su correspondiente acción. Con los árboles de decisión es difícil pasar por alto una condición importante dentro del diseño.

En las empresas de tipo adminsitrativo (y aún en las que no lo son), las acciones frecuentemente están condicionadas a una serie de requerimientos. Por ejemplo: en un sistema de almacén de una refaccionaria X una pieza puede darse de baja por diversos motivos, venta directa al cliente, transpaso a otros almacenes de la red, robo, venta/transpaso al departamento de servicios al cliente, transpaso a una unidad perteneciente a la empresa, etc. Todas estas ramas pueden hacer bastante difícil establecer las acciones a seguir. Como si la labor del analista no terminara allí, las acciones a tomar dependen en una buena parte de las *políticas* de la empresa en cuanto a la forma como se manejan diversos asuntos administrativos. Las políticas de la empresa son las directrices que guían el comportamiento en diversas circunstancias. De tal forma que una pieza de dicho almacén puede ser registrada como un transpaso cuando en realidad se efectuó una venta con un descuento tal que la pieza fue expedida con el valor de costo.

Los instrumentadores de las políticas en las empresas deben conocer las decisiones que los sistemas de automatización de información tomarán. Es difícil platicar con un directivo o un gerente y, si no se dispone de herramientas gráficas que permitan plantear fácilmente los que el sistema realizará, se corre el grave riesgo de mal interpretar la información de lo que el sistema realmente debería de hacer.

El árbol de decisión muestra de un vistazo que condiciones son más relevantes que otras. Fuerza al analista a establecer de manera clara y sin ambiguedades, la secuencia de decisiones que llevan a la ejecución de un proceso determinado.



Una ventaja adicional de los árboles de decisión es que nos presentan en forma explícita algunos de los requerimientos de información más importantes. Los datos que son de fundamental importancia para una toma de decisión aparecen enunciados en los nodos del árbol. Es importante recalcar que algunos de los datos importantes podrían encontrarse implícitos bajo un nombre genérico. En un nodo podríamos encontrar que una factura es importante para dar como válida una reclamación del cliente, también es importante que la factura contenga la fecha de venta, nombre del cliente, etc. Dichos datos podrán no aparecer en forma explícita en el nodo de decisión, pero el analista debe ser capaz de reconocerlos como elementos importantes de información.

Los árboles de decisión rara vez se utilizan solos, no son un método sino una herramienta de análisis. Además no siempre son el medio más adecuado de establecer las decisiones que es necesario tomar dentro de los sistemas. En forma intuitiva se capta que un sistema muy grande genera árboles con múltiples ramificaciones que pueden llegar a convertirse en algo demasiado complejo como para que se considere que el árbol de decisión resulta de verdadera ayuda. Para estos casos se suele utilizar una herramienta hermana del árbol de decisión que es la **tabla decisión.**

TABLAS DE DECISION

Un método más formal para el planteamiento de las decisiones es la Tabla de decisión. Un analista recibe una tremenda ayuda cuando utiliza esta herramienta que le permite manejar las decisiones de una manera independiente a la secuencia en la cual serán tomadas.

Una tabla de decisión es una matriz de renglones y columnas que relacionan las condiciones que se deben de cumplir para poder realizar una determinada acción. Existen diversos formatos para representar una tabla de decisión. Iniciaremos el estudio de las tablas de decisión con el formato limitado y un ejemplo corto.

CONDICIONES	REGLAS DE DECISIÓN
IDENTIFICADOR DE CONDICIONES	ENTRADAS DE ACCIONES
IDENTIFICACION DE LAS ACCIONES	ENTRADAS DE LAS CONDICIONES

En un sistema de control escolar las inscripciones al laboratorio de electricidad II están sujetas a que el alumno sea regular en la seriación de dicha materia (haber aprobado electricidad I) . Se debe presentar un comprobante de pago, excepto para aquellos que estén becados o que hallan aprobado electrónica I con un promedio superior al 9. ¿En qué casos el alumno debe pagar la inscripción? ¿En qué casos se le acepta como alumno inscrito?

CONDICIONES	REGLAS DE DECISION							
C1 Alumno regular	SI	SI	SI	SI	NO	NO	NO	NO
C2 Comprobante de pago	SI	SI	NO	NO	SI	SI	NO	NO
C3 Promedio superior al 9	SI	NO	SI	NO	SI	NO	SI	NO
A1 Alumno se inscribe	✓	✓	✓					
A2 Pagar		✓		✓				

La tabla de decisiones del ejemplo, posee en la parte superior izquierda, el conjunto de condiciones que se tienen que identificar para poder establecer si una acción será ejecutado o no. Una condición solo puede tener dos valores: cierto (SI) o falso (NO). Los valores que dichas condiciones pueden adquirir son mostrados en la parte superior derecha. En esta zona se muestran de manera inicial todas las posibles condiciones que pueden tomar las condiciones.

En la parte inferior izquierda se identifican las acciones que el sistema tendrá que efectuar en base a las condiciones mostradas. Obsérvese que aunque un alumno necesite pagar antes de inscribirse, el orden en que esto es realizado no es prioritario para establecer la acción que se desarrollará.

En la parte inferior derecha se encuentra la zona que relaciona de manera unívoca las relaciones con las acciones. Para el caso de estudio se aprecia que un alumno debe pagar solamente cuando es un alumno regular y no tenga un promedio superior al nueve .

La metodología para implantar estas funciones en un programa quedará más clara en el epígrafe II.2.3 de estos apuntes.

CONSTRUCCIÓN DE LAS TABLAS DE DECISIÓN

Para poder construir una tabla de decisión, un programador o un analista debe seguir la serie de pasos que se listan a continuación:

1.- **Identificar las condiciones:** El analista debe identificar cuales son los factores más relevantes para los procesos de decisión. Tal y como se mencionó en párrafos anteriores, las condiciones deben de poder responderse con un SI o un NO. Esto permitirá su manipulación dentro de la tabla, así como aplicar métodos avanzados de reducción algebraica.

2.- **Identificar las actividades:** Los procesos dependerán de las condiciones encontradas en el punto anterior. Es importante que el analista determine claramente cuales son las condiciones que tienen que cambiar para ejecutar o no un proceso determinado. Esta actividad suele ser un proceso de continua realimentación con los usuarios del sistema a ser implantado. Generalmente ellos no consideran cosas importantes por la sencilla razón de que por ser *cosas sabidas* no les parecen trascendentes.

3.- **Analizar las 2^n combinaciones posibles.** Siendo n el número de condiciones encontradas en el punto uno. En el ejemplo se tienen tres condiciones y ocho combinaciones posibles (2^3=8). Para cada combinación debe establecerse con claridad si se ejecuta un proceso dado.

4.- **Llenar la tabla con las reglas de decisión:** Para realizar ésto se obtiene el número de combinaciones posibles según se estableció en el punto anterior. Con la primera condición se comienzan a llenar los renglones de condición con un **SI** y un **NO** alternados hasta llenar el número de combinaciones. Con la segunda condición se llena el renglón correspondiente con dos **SI** y dos **NO** en forma alternada. El tercer renglón de condición se llenará con cuatro **SI** y cuatro **NO** en forma alternada. Para el renglón n se llenará con 2^{n-1} **SI** y 2^{n-1} **NO**.

En general la tabla de decisión no ayuda a encontrar redundancia en cuanto al cumplimiento de determinadas condiciones. En el ejemplo se vió que aunque existen dos combinaciones para que un alumno pague su laboratorio, las tres condiciones no eran trascendentes ya que solo importaba que el alumno fuera regular y que tuviera un promedio inferior al nueve.

Las tablas de decisión también eliminan las posibles contradicciones que pudiesen surgir durante la etapa de análisis de requerimientos.

I.1.4 Instrumentación de los módulos

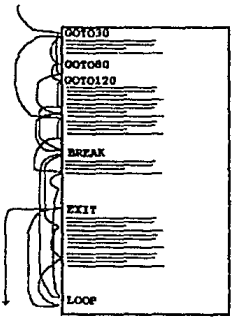
ANTECEDENTES

El diseño de los módulos y las características externas, así como algunas de las características que los conforman han sido hasta éste momento la parte medular de estos apuntes. Se han plantedo algunos fundamentos básicos de lo que se conoce como *Análisis Estructurado de Sistemas* y aunque el tópico abarcaría una obra mucho mayor a la presentada en éstas notas, se ha hecho énfasis en aquellos temas que nos conducen a un buen diseño modular. Es momento de descender un nivel de abstracción y enfocarnos ahora a la construcción de los módulos.

Se han comentado las características más importantes de los módulos vistos como un todo. Cohesión, acoplamiento, tablas de decisión y árboles. Es momento de ver un módulo en sus partes constitutivas, es decir, las estructuras de control que lo conforman.

Después del surgimiento de las primeras computadoras, fueron muy pocas las empresas que podían darse el lujo de tener un equipo tan costoso. Prácticamente su uso se restringió a las empresas gubernamentales en donde el volúmen de información justificaba el tremendo gasto que un equipo de enormes dimensiones representaba. Durante este periodo se comenzaron a desarrollar sistemas cuya complejidad se fué haciendo cada vez mayor. Surgieron los sistemas operativos y los primeros compiladores para lenguajes de alto nivel. Históricamente se considera que el compilador de FORTRAN desarrollado en 1957 por IBM es el primer lenguaje que goza de una gran popularidad. Este lenguaje es en sus inicios bastante amable con los programadores, pero carece de una disciplina en sus instrucciones que da lugar a programas con *arquitectura de espagetti*. FORTRAN se establece como el lenguaje de programación nativo para muchos equipos.

En verdad que FORTRAN no es el único lenguaje de alto nivel que surge. ALGOL es más disciplinado y surgió inclusive antes que FORTRAN, pero sus índices de popularidad nunca llegaron a equipararse. Aquí, como en muchas otras ramas del saber, lo popular no siempre es lo mejor. Para cuando los equipos de cómputo comenzaron a volverse más populares se tuvo una gran demanda de personal que programase dichos equipos. Generalmente el personal más calificado se encontraba en la universidades desarrollando los sistemas operativos interactivos, de aplicación distribuida, de tiempo real, compiladores. etc. El resto de los mortales tuvo necesidad de improvisar. Los sistemas de cómputo costaban mucho dinero y los programas salían hechos rápidamente. Lo importante era programar, no importaba gran cosa como quedaran estructurados internamente. La *arquitectura de espagetti* o de sopa de fideos surgió como la manera más normal de realizar un programa.



ARQUITECTURA DE ESPAGETTI

Surgieron los sistemas distribuidos, de tiempo compartido y el tamaño de las aplicaciones creció a dimensiones que aún ahora resultan difíciles de comprender. Programas de 100' 000 líneas de código comenzaron a ser cosa bastante común para ciertas empresas cuyo campo de acción es el mundo entero.

Y en medio de todo ésto, los sistemas de software comenzaron a reclamar mantenimiento. De pronto se vió la urgente necesidad de encontrar técnicas y métodos que permitieran modificar rápidamente los programas creados, que redujeran costos de mantenimiento, depuración, actualización, etc. Los programas debían ser más flexibles de lo que anteriormente eran. En fin, el costo de los programas comenzó a crecer de una manera tan alarmante que era inclusive mayor al del equipo de hardware.

Ante esta situación, algunos estudiosos comenzaron a buscar soluciones a dichos problemas. Edsger W. Dijsktra en 1965 publicó sus *Notas sobre Programación Estructurada* en donde expone algunas de sus reflexiones sobre los programas. Años más tarde publicó *Técnicas de Ingeniería de Software*. Sin embargo, su voz no es muy escuchada más que en ciertos círculos académicos. En general, la tendencia se encontraba centrada en el estudio de las estructuras de datos más que en los programas en sí.

En la década de los 70's, el francés Jean-Dominique Warnier desarrolló la *programación lógica* la cual expuso en numerosas obras. Orr añadió algunos puntos de vista a los conceptos presentados por Warnier y en conjunto publicaron *Desarrollo de Sistemas Estructurados*. Sin embargo, es necesario hacer gran hincapié en que Warnier es frecuentemente citado en los libros de estructuración de programas pero él no estaba convencido de la validez de la programación estructurada. Warnier definitivamente tenía un claro enfoque orientado a los datos y los diagramas Warnier-Orr son diagramas orientados a los datos no a los programas.

En 1975 Edward Yourdon y Larry Constantine publican *Técnicas de estructuras de programas y diseño* que sería el primero de una larga lista de libros que tratan del mismo tema y que dieron fundamento a la creación del Análisis Estructurado. Según comentan los autores en dicha obra, para 1975 la NASA y otras instituciones mundialmente conocidas aún continuaban utilizando técnicas no estructuradas para la creación del software que empleaban.

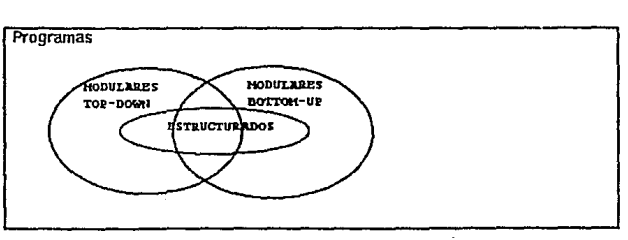
OBJETIVOS

Aunque han pasado muchos años desde entonces, aún sigue malinterpretándose el verdadero objetivo de la programación estructurada. Algunos la reducen simplemente a la eliminación del salto incondicional de los programas como estructura de control. Muchos más aún reducen el objetivo a la simple eliminación del GOTO en la codificación. En su lugar emplean como sustituto sentencias equivalentes pero que no suenen a un salto incondicional aunque en el fondo lo sean : LOOP, EXIT, BREAK, etc.

La programación estructurada está enfocada a las estructuras de control de un programa. Su técnica primaria consiste en la eliminación del salto incondicional y su reemplazo por sentencias bien estructuradas de bifurcación y control.

La programación estructurada es un caso especial de la programación modular. El diseño de un programa estructurado se realiza construyendo bloques tan pequeños que puedan ser codificados fácilmente y esto se logra hasta que se alcanza el nivel de *módulos atómicos*, es decir, sentencias individuales (SI-ENTONCES , HAZ MIENTRAS, etc).

Un programa modular puede seguir todas las reglas que hemos expuesto hasta este momento, pero internamente ser tan inestructurado como el que más



La programación estructurada en el universo de los programas

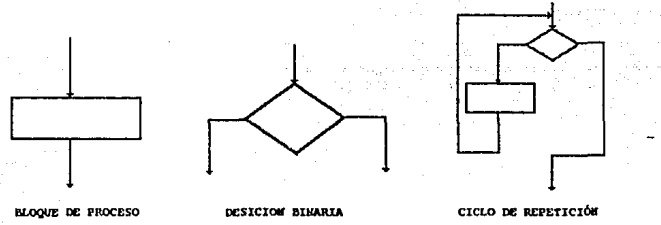
La definición de la programación estructurada exige de estructuras de control del programa que sean *bien estructuras.* La pregunta que surge es: ¿Cuáles son las características de una estructura de control bien definida?

La contestación a ésta pregunta viene dada por los dos siguientes principios básicos.

TEOREMA DE LA ESTRUCTURA

El teorema de la estructura fué enunciado por Bohm y Jacopini. En él se establece que se requieren de tres bloques básicos para construir cualquier programa:

1.- Una caja de proceso
2.- Una decisión binaria
3.- Un mecanismo de repetición

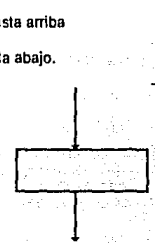


El enunciado fué hecho en el artículo *Diagramas de flujo, Máquinas de Turing y Lenguajes con solo dos reglas de formulación* publicado en italiano en 1965

PROGRAMA PROPIO

Se llama programa propio a aquel que cumple con los siguientes requisitos:

- Tiene un solo punto de entrada hasta arriba
- Se lee de arriba hacia abajo
- Tiene un solo punto de salida hasta abajo.



Recapitulemos. Se había llegado a la conclusión de que una máquina de Turing podía considerase como una máquina capaz de tomar decisiones. También se mencionó la factibilidad de elaborar en base a la máquina de Turing, un modelo de modelos y que dicha máquina es lo que actualmente conocemos como una computadora.

Alan Turing planteó su máquina como una hipótesis puramonte formal y matemática. A su vez Bohm y Jacopini exponen con argumentos puramente matemáticos el *Teorema de la estructura* en el cual demuestran que una máquina de Turing basta que disponga de tres estructuras de control para poder representar cualquier algoritmo. Las implicaciones que ésto tiene para la programación son enormes.

Si un algortimo para la máquina de Turing solo requiere de tres estructuras de control, significa que un compilador puede ser terriblemente pequeño o que la sintaxis de los lenguajes puede reducirse a un mínimo. También implica que un programador debe conocer sólo un conjunto finito y muy reducido de sentencias de control para implantar *cualquier* algortimo.

En fechas recientes esto ha marcado las tendencias en los mismos microprocesadores con los cuales se fabrican las computadoras. La tecnología RISC aboga por un conjunto muy reducido de instrucciones con las cuales se puede programar cualquier cosa, en contra de los microprocesadores con una cantidad inmensa de instrucciones tan especializadas que se convierten en prisiones poco flexibles para el programador.

Esta tendencia tiene su razón de ser eminentemente histórica y regida por las necesidades de los programadores. En los albores de la computación los compiladores y los lenguajes eran diseñados en base a los microprocesadores existentes. Ahora, los microprocesadores se diseñan pensando en los lenguajes y en las necesidades de software existentes en el mercado. Los programadores dictan las necesidades de hardware y sobre eso se efectúa el diseño.

La existencia de un conjunto finito de sentencia de control nos permite estandarizar los procesos internos de los programas. La creación individual sigue siendo tan válida como lo ha sido siempre, pero al existir un conjunto estándar de sentencias en los programas la lectura de los mismo se torna más sencilla. El programador solo tendrá que leer estructuras a las cuales está acostumbrado.

Por otra parte, la típica arquitectura de sopa de fideos prevaleciente en la mente de muchos programadores desaparece con todo el mar de confusión que esto provocaba.

El estilo de programación se toma más elegante y disciplinado. Es necesario reconocer que programar estructuradamente puede llegar a costar bastante trabajo extra para una persona que no lo ha hecho nunca de esta manera, pero para un programador iniciado pronto se convierte en algo tan normal como el respirar.

Y para programar estructuradamente solo es necesario pensar hasta donde sea necesario con el pensamiento modular introducido en los primeros temas.

COSA ABSURDAS QUE NO LO SON TANTO

Después de la definitiva sorpresa que causa el Teorema de la estructura encontramos la definición de programa propio. *¿Que, qué?* me han dicho algunos alumnos "*Ese concepto suena definitivamente tonto.*" Y de hecho es realmente un concepto tonto. Todo programa debería de comenzar en la primera línea, continuar con las que siguen y finalizar con la última para salir por abajo... Bueno así debería ser en la vida real, pero como todas las cosas ideales, en la vida real esto no ocurre así.

El concepto de programa propio es una equivalente a lo que la Ley cero es a la termodinámica. Es una ley tan evidente que a nadie se le había ocurrido pensar en que existía, pero existe y rige los procesos termodinámicos. El concepto de programa propio es tan evidente... que pocos programadores profesionales se preocupan por él.

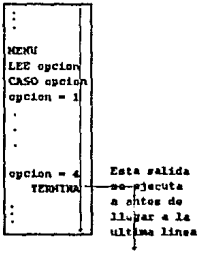
De buenas a primeras resulta difícil pensar en un ejemplo que no cumpla esta *ley evidente* . En el transcurso de este capítulo se ha presentado, por lo menos, un ejemplo gráfico de dicho caso. En el capítulo dos se muestran técnicas para lograr programas propios aúnque parezca difícil a primera vista.

El pretexto más común para violar este concepto lo encontramos en los programadores que gustan de la instrucción `exit()` y `return()` como solución a circuntancias específicas y algunas de estas *situaciones específicas* son muy comunes. Dichas instrucciones provocan rompimientos incondicionales de la secuencia de instrucciones y retoman a los programas padre que han generado los módulos hijo desde donde se ejecutan dichas sentencias. El primer ejemplo lo tenemos en la programación del menú principal de un sistema:

```
PROGRAMA PRINCIPAL

INICIALIZACION DE VARIABLES
.
.
.
PINTA MENU:
      1.- ALTAS
      2.- BAJAS
      3.- CAMBIOS
      4.- SALIDA
LEE opcion
```

```
HAZ SEGUN CASO DE opcion
      CASO DE opcion = 1
            LLAMA ALTAS
      CASO DE opcion = 2
            LLAMA BAJAS
      CASO DE opcion = 3
            LLAMA CAMBIOS
      CASO DE opcion = 4
            TERMINA EJECUCION
FIN CASO
.
.
.
```

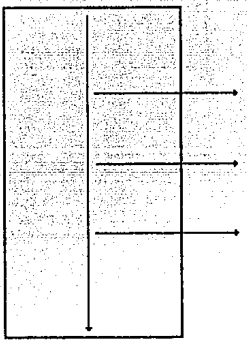
En el caso presentado se despliega en la pantalla de terminal un menú de cuatro opciones. Para las tres primeras opciones se lee la opción elegida por el usuario y posteriormente se llama a la rutina responsable de dicho proceso. Pero en el cuarto caso no se llama a una rutina que cierre archivos, libere memoria o cosa así. Simplemente se coloca un `exit()` que da al traste con el concepto de programa propio para este programa



Quisiera creer que ésta es una situación poco común, pero bajo las actuales tendencias del software creo que esto se seguirá presentando en más de un programa. Aquí se particularizó el ejemplo para el programa principal, pero en la gran mayoría de las ocasiones esto ocurre desde un módulo hijo en algún sub submenu en un sistema cualquiera. En el mejor de los casos no se encuentra una sentencia de salida de programa, sino de retorno al programa padre. En realidad esto cambia la situación en muy poco.

El segundo caso típico es terriblemente preocupante. De hecho es lo que hace abogar por la escritura de programas propios y no por su contraparte. El caso en cuestión es el relacionado con el manejo de los errores en un sistema. Veamos el siguiente pseudocódigo

```
Modulo XYZ

INICIALIZACION DEL MODULO
.
.
.
SI (existe archivo_maestro) ENTONCES
      ABRE archivo_maestro
SINO
      TERMINA PROGRAMA
FIN DEL SI

HAZ MIENTRAS (no se procesen todos los empleados)
      TOMA campo_llave
      SI (campo_llave es invalido)
            REGRESA AL PROGRAMA PADRE
      FIN DEL SI
      TOMA sueldo
      SI (sueldo es nulo)
            DESPLIEGA ERROR
            TERMINA PROGRAMA
      FIN DEL SI
      CALCULA PAGO
FIN HAZ
CIERRA archivo_maestro
.
.
.
```



El programa tiene salidas que no necesariamente se encuentran en la parte de abajo

Este caso aparenta tener estructuras bien definidas. Sin embargo tiene más de una salida. Cada una de las salidas está diseminada por todo lo largo y ancho del programa fuente. No se trata de postular un par de teoremas y obligar a la gente a utilizarlos a fuerza de golpes. La importancia del programa propio se hace evidente en éste ejemplo.

Suponga que un buen día un usuario tiene un problema con el sistema y le manda llamar para que solucione el problema. El problema reside en algún punto extraño del sistema, ya que cuando se pide la opción del calculo de nómina, el programa aborta abruptamente y retorna el control al sistema operativo. El usuario está realmente extrañado. Eso no debió suceder, amén de que se perdió todo su trabajo de captura de las incidencias de su nómina. Por otra parte, usted con esa sonrisa habitual de quien tiene todo bajo control toma asiento frente a la terminal y es entonces cuando descubre que la rutina pudo haber salido en 2, 3, o múltiples puntos distintos. ¿Por dónde se debe iniciar a buscar el error? ¡Lástima de sonrisa! ¿No cree? Ahora tiene que averiguar si es el módulo XYZ el culpable en alguna de sus salidas a sistema operativo, o bien, si la única salida de retorno al módulo padre conmuta el control y es en el módulo padre en donde ocurre el error. ¡Imagine la sensación de frustración cuando la arquitectura mostrada en este módulo se repite en todos y cada uno de los 80 módulos de los que consta el sistema !

La existencia de múltiples puntos de salida o de salidas con distintos comportamientos (salir al sistema operativo o al programa padre) redundará en programas de muy difícil mantenimiento. Existen muchos programadores que arguyen que en ocasiones no se tiene de otra más que tomar el camino corto para corregir un error que ha surgido de pronto en un módulo clave. Dicen que en ocasiones es más eficiente programar un salto incondicional que viole el principio del programa propio. Pero la verdad sea dicha , no he encontrado un sólo caso en que ésto suceda.

El capítulo dos muestra algunas de las alternativas existentes para evitar esta multiplicidad de salidas o *la necesidad* de un salto incondicional. Aunque no es un tratado exhaustivo, sí aporta bastantes ideas. Arriesgarse a violar alguna de las reglas del buen estilo en el programar es arriegarse a que se cumpla la ley de Murphy

Todo lo que pueda fallar, fallará

ESTRUCTURAS ESTRUCTURADAS DE CONTROL

Los dos conceptos expuestos en la discusión anterior no llevan a la búsqueda de un conjunto de estructuras que cumplan en forma cabal ambas definiciones. Aunque en realidad el Teorema de la estructura no deja lugar a ningún tipo de busqueda.

Con el Teorema de la estructura y el concepto de Programa propio se construyen todas las estructuras sintácticamente válidas para esta filosofía de programación. Por una parte el salto incondicional debe estar eliminado como una forma de programación, por el otro tenemos que cualquier estructura que se nos ocurra elaborar debe estar construida sólo por los tres elementos básicos mencionados en el Teorema de la estructura, por si esto fuera poco, dichas estructuras deben cumplir con el concepto de programa propio.

¿Algo más? ¡ Pues sí!. Las estructuras resultantes deben tener la facultad de ser concebidas como una extensión del pensamiento modular que se ha analizado hasta éste momento.

Las estructuras que cumplen con dichas condiciones son presentadas a continuación. Debe recordarse en todo momento que se trata de un conjunto de estructuras elementales que en forma ideal se encuentran en cualquier lenguaje. Son las formas más comunes de control.

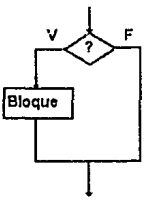
EL BLOQUE DE PROCESO

El bloque de proceso es generalmente interpretado como la estructura que se ejecuta en secuencia. Es decir, es necesario que los programas sean ejecutados en forma secuencial, instrucción por instrucción, para que el programa pueda ser leído de arriba a abajo. En general los lenguajes de programación son de ejecución secuencial. Salvo aquellos casos de programación paralela en los cuales más de una instrucción se ejecuta a la vez. Aunque en forma estricta el programa sigue leyéndose de arriba a abajo.

Una característica esencial de éste bloque de proceso es que no necesariamente puede ser una sola sentencia. Pueden ser un conjunto de sentencias cuya ejecución sea secuencial. Las sentencias puederi ser o un bloque de sentencias o cualquiera de las sentencias que a continuación serán presentadas.

DECISION BINARIA

La estructura de decisión binaria más sencilla es la decisión con una sola rama. A ésta estructura se le conoce como SI - ENTONCES (IF - THEN) . En ella se pregunta por el valor lógico de una condición predeterminada. Si la condición es evaluada con el valor de verdadero entonces se ejecuta un bloque de proceso. En caso de que la condición sea evaluada como falsa entonces no se ejecuta ninguna de las sentencias incluidas en la rama verdadera de la estructura y se continúa con la ejecución de la sentencia ubicada inmediatamente después del fin de la condición.

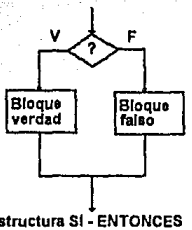


Decisión binaria de rama verdadera

El pseudocódigo para esta estructura ha sido manejado extensamente a lo largo de estos apuntes

```
SI (condición es verdadera) ENTONCES
      EJECUTA BLOQUE DE PROCESO
FIN DE SI
```

La variante es la decisión binaria de dos ramas. En ella es evaluada una condición y si la condición es evaluada como verdadera entonces se ejecuta el bloque correspondiente a la rama verdadera. Al alcanzar el fin de este bloque, la siguiente instrucción que se ejecutará será la que se encuentre inmediatamente después del cierre de estructura (FIN DEL SI). En caso de que la condición sea evaluada como falsa entonces no se ejecutará ninguna de las instrucciones contenidas dentro de la rama verdadera de la estructura y se procede a la ejecución de la rama falsa de la estructura. Al igual que sucede con la rama verdadera, al llegar a la última sentencia del bloque falso, se ejecutará enseguida la instrucción ubicada inmediatamente después del fin de estructura.



Estructura SI - ENTONCES

El pseudocódigo de esta estructura es el siguiente:

```
SI ( condición es verdadera ) ENTONCES
      EJECUTA bloque verdadero
SINO
      EJECUTA bloque falso
FIN DEL SI
```

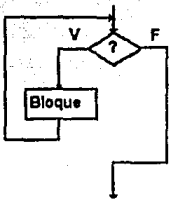
ESTRUCTURAS DE REPETICIÓN

En base a la estructura de decisión podemos construir los ciclos de repetición. Estrictamente hablando, sólo es necesario un mecanismo de repetición en un lenguaje para que éste sea completo y suficiente para instrumentar cualquier algortimo. El mecanismo de repetición tiene el nombre inglés de **DO-WHILE** (HAZ-MIENTRAS).

El funcionamiento del mecanismo de repetición es el siguiente: El inicio de la estructura indica una decisión que definirá si el proceso es ejecutado o no. Si la condición que se evalúa a la entrada con el valor de verdadero, entonces se procede a la ejecución del bloque de instrucciones definido dentro del cuerpo del ciclo. Tan pronto como es alcanzada la última instrucción del bloque de instrucciones se vuelve a evaluar la condición a la entrada de la estructura. Si la condición aún es evaluada con valor de verdadero se repite la ejecución de las sentencias que conforman el bloque de proceso del while.

Si la condición a la entrada del while es evaluada con el valor de falso no se ejecuta ninguna de las instrucciones que conforman el cuerpo del while y se procede a la ejecución de aquella instrucción que se encuentre inmediatamente despúes del cierre de estructura.

El diagrama de flujo muestra la representación gráfica de la estructura.



Estructura HAZ - MIENTRAS

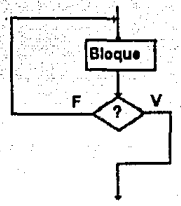
El pseudocódigo correspondiente es el siguiente:

```
HAZ MIENTRAS ( condición sea veradera )
      EJECUTA bloque
FIN DEL HAZ
```

Una característica muy importante de esta estructura, lo consistuye el hecho de que el bloque de instrucciones contenido en el cuerpo del ciclo puede llegar a no ejecutarse ni una sola vez. Si la condición que da entrada al ciclo resultara evaluada como falsa nos encontraríamos con que dicho bloque no sería ejecutado. Esto tiene una trascendencia en el momento de escribir el código de un ciclo while. Las variables de control del ciclo while deben poseer valores previos que garanticen la entrada al ciclo. Además, dichas variables deben modificar sus valores dentro del cuerpo del ciclo del while, ya que de no suceder así, el ciclo se ejecutaría infinidad de veces y se requeriría de una intervención violenta por parte del usuario (digitar una secuencia de teclas que rompan la ejecución del programa), o bién el uso del salto incondicional que tan sancionado ha resultado a lo largo de estos apuntes. Un ciclo while bien estructurado contendrá una condición clara que indique cual es la condición que romperá con la repetición.

Una estructura tan popular como el ciclo while es el **repeat - until** (REPITE - HASTA). En sí misma es muy parecida a la estructura while analizada hace un momento. Consiste de un bloque de instrucciones que se ejecutará en forma secuencial y que conformarán el cuerpo del ciclo. Tan pronto como se alcance el fin de bloque se realizará la evaluación de una condición. Si la condición resultara evaluada como falsa, entonces se procederá a la repetición del bloque de instrucciones. En caso de que la condición en cuestión resultara evaluada como verdadera, se dará por concluída la labor de repetición y se continuará la ejecución del programa con la instrucción que se encuentre ubicada inmediatamente despues del cierre de estructura.

El diagrama de flujo muestra la forma en que se ejecutan las instrucciones dentro de esta estructura:



Estructura REPITE - HASTA

Y su pseudocódigo es el mostrado a continuación:

```
REPITE
      EJECUTA bloque de instrucciones
HASTA ( condición sea verdadera )
```

A diferencia del do-while, en la estructura **repeat - until** se tiene que el bloque instrucciones se ejecutará por lo menos una vez independientemente de los valores que puedan tener las variables que controlan la ejecución del ciclo. La condición que controla el ciclo debe además, evaluar como falsa para poder realizar una repetición del bloque de instrucciones, totalmente contrario a lo que sucede con el ciclo while en la cual la condición debe ser evaluada como verdadera.

En general los lenguajes de programación implantan el do - while como una estructura de control más común a lo que es el repeat - until. Cada una de estas estructuras son valiosas para determinadas circunstancias. En general un repeat-until puede sustituirse con un do-while y viceversa excepto en aquellos casos en los cuales el número de repeticiones varía de cero a varias veces. Esta particularidad nos hace considerar al do -while como una estructura más fundamental a lo que lo es el repeat - until.

Un ciclo controlado por variable (for) es un caso particular de un do - while y por tanto no se considera como estructura fundamental ya que puede construirse a partir de una forma más elemental.

LA SELECCION MULTIPLE

Las estructuras de control fundamentales han sido estudiadas ya en los apartados anteriores. A partir de ellas se puede construir un superconjunto de estructuras no fundamentales y aún validas para la programación estructurada.

La selección múltiple es necesaria en aquellos casos en los cuales se requiere preguntar por el valor que tomará una variable. El conjunto de valores que dicha variable puede poseer es finito y predictible. La estructura de selección múltiple es construída a través de estructuras de selección binaria anidadas.

La selección múltiple es conocida también como estructura **case** (case : caso). En los diagramas mostrados se presenta la forma de elaborar la estructura con la decisión binaria, así como su representación en diagrama de flujo.

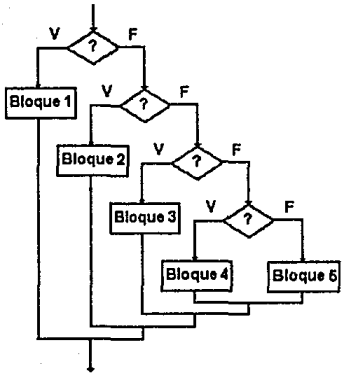


DIAGRAMA BASICO DE LA SELECCION MULTIPLE

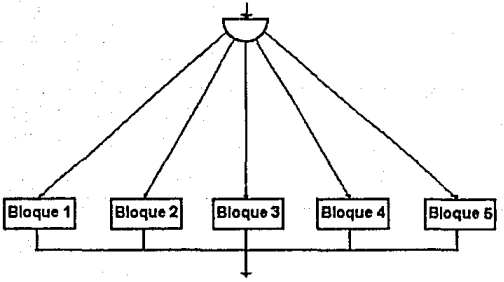


DIAGRAMA DE FLUJO DE LA SELECCION MULTIPLE

ESTRUCTURAS DE CONTROL, PROGRAMA PROPIO Y MODULARIDAD

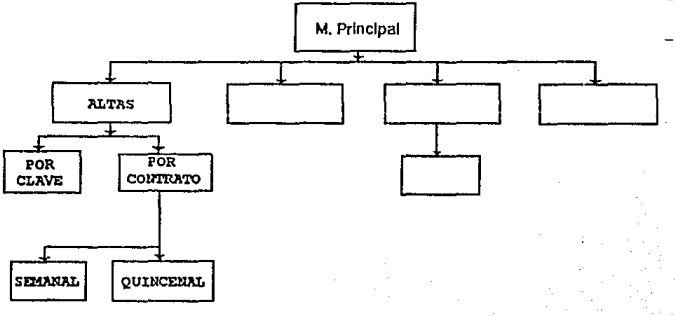
La implantación de los módulos ha quedado restringida por el Teorema de la estructura al uso exclusivo de las tres estructuras básicas de control. Cualquier otra estructura que deseemos construir debe estar formada por esas tres construcciones básicas. El salto incondicional no aparece mencionado en ninguna parte. Las estructuras que hemos presentado cumplen con estas características.

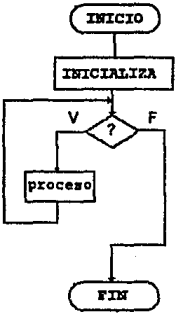
También se había establecido como una premisa de la construcción de dichas estructuras, que cada bloque de control debería cumplir con el concepto del programa propio. La importancia de este punto es vital para la construcción de los programas.

En este momento tenemos los elementos de un mecano con el cual podemos construir una infinidad de modelos distintos, la única condición para que las piezas del mecano ensamblen correctamente es que se puedan unir entre sí todas las piezas con todas las piezas. Cada estructura tiene un sólo punto de entrada hasta arriba y una sola salida hasta abajo. En forma de secuencia ya se les puede unir para formar un *collar de estructuras* que enlazados pueden formar un programa. Observando más detenidamente a todas las estructuras, encontramos un detalle aún más importante. En todas las estructuras encontramos un *bloque de instrucciones* que ha sido manejado en su forma más abstracta. Hasta ahora no nos ha importado averiguar qué es lo que ese bloque contiene.

Cada bloque de instrucciones estará conformado por las mismas estructuras que se han estudiado. Esto es consistente con la discusión que se hizo sobre los grados de abstracción necesarios para la implantación de los módulos.

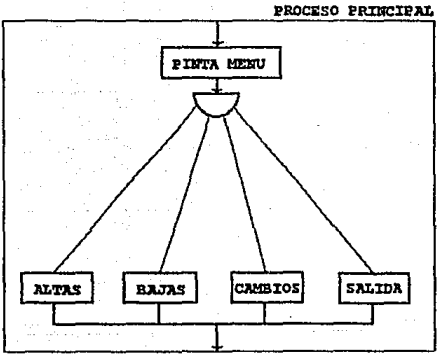
Obsérvese el siguiente ejemplo. Se trata de un sistema cualquiera en el cual existen básicamente 4 módulos principales. En el nivel uno diseñamos el pseudocódigo para abarcar exclusivamente estos módulos. Dentro del las estructuras que controlarán éstos módulos sólo se indica que ejecutarán una serie de instrucciones no definidas pero que sólo contienen una entrada y una sola salida. En el nivel dos encontramos que el módulo de altas está conformado por las estructuras de control ya conocidas. El gráfico muestra que uno de los módulos de la estructura de selección múltiple es una vez más una estructura elemental.





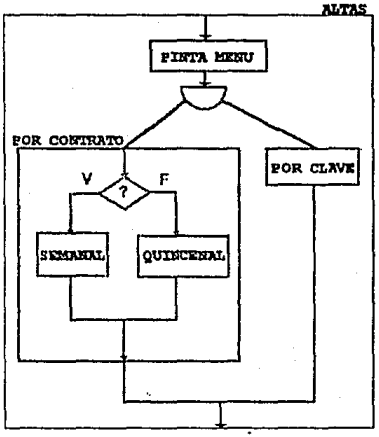

```
MODULO PRINCIPAL
   INICIALIZA VARIABLES
   HAZ MIENTRAS (se desee ejecutar)
       EJECUTA proceso principal
   FIN DEL HAZ
```

```
PROCESO PRINCIPAL

PINTA MENU
   1.- ALTAS
   2.- BAJAS
   3.- CAMBIOS
   4.- SALIDA
LEE opcion
HAZ CASO DE opcion
    CASO opcion = 1
        LLAMA ALTAS
    CASO opcion = 2
        LLAMA BAJAS
    CASO opcion = 3
        LLAMA CAMBIOS
    CASO opcion = 4
        LLAMA SALIDA
FIN CASO
```



```
ALTAS
PINTA MENU
  1.- POR CONTRATO
  2.- POR CLAVE
LEE opcion
HAZ CASO DE opcion
   CASO DE opcion = 1
        EJECUTA ALTAS POR CONTRATO
   CASO DE opcion = 2
        EJECUTA ALTAS POR CLAVE
FIN DE CASO
```

```
ALTAS POR CONTRATO
SI (contrato es semanal) ENTONCES
    LLAMA ALTAS POR SEMANA
SINO
    LLAMA ALTAS POR QUINCENA
FIN DEL SI
```



Esta repetición de las estructuras a nivel detalle dentro de los programas es lo que da poderío a la filosofía de la programación estructurada como un enfoque de construcción de programas. *Con solo tres herramientas básicas, matemáticamente suficientes para una máquina de Turing, es posible construir cualquier problema decible y por tanto elaborar un modelo válido y predictible de un problema real.*

La abstracción lograda de las estructuras de control las hace independientes de cualquier lenguaje de programación. Es más, la filosofía de la programación estructurada no niega su inclusión dentro de macrofilosofías que hagan uso de los conceptos que la fundamentan. No está orientada a los datos y por tanto el manejo que se pueda hacer en la represenación de estos no influye esencialmente en la arquitectura de construcción de los programas.

Hemos desarrollado la fórmula básica para la escritura de buenos programas con un estilo estructurado y disciplinado.

Preguntas

1.- Escriba un ensayo sobre:

1.1 Las caracteristicas de un buen programador

a) Para escribir este ensayo plantee el punto de vista de la empresa, el punto de vista de un programador y el punto de vista de un usuario.

b) ¿Cuáles de estas caracteristicas pueden ser aprendidas?

c) ¿Un buen programador nace o se hace?

d) ¿La experiencia es un factor decisivo para que una persona sea un buen programador?

e) Si considera que es posible, ¿Cómo puede cualquier persona convertirse en un buen programador?

1.2 El impacto de los computadores en el mundo moderno

a) ¿Cuáles han sido los factores que han impulsado el auge de los computadores?

b) ¿Qué papel juega la tecnología y que papel juega el software?

c) ¿Cómo se ve afectado el hombre de la calle con las computadoras?

d) ¿Cuál es el papel del ingeniero en computación en todo éste marco?

1.3 Los métodos de diseño en las empresas

Para escribir este ensayo investigue en diversas empresas los métodos de diseño para los módulos

a) ¿Están estandarizados los métodos en las empresas?

b) ¿Los módulos se programan en base a los conceptos que se expusieron en el presente capítulo?

c) ¿Existen muchas empresas que no aplican una metodología en el diseño y la documentación?

d) Si usted fuera jefe de sistemas de alguna empresa. ¿Cómo integraría a su equipo de trabajo para seguir una metodología?

e) Exponga sus conclusiones en forma clara

2.- Explique porque un programa es la mitad de una computadora

3.- Con sus propias palabras explique como podría llegar a medirse un programa. Si considera que esto no es posible, explique que hace falta para que sea posible

4.- ¿Cuál es la clave del diseño en el software?

5.- Busque en tres diccionarios el significado de la palabra abstracción y apliquelos a la creación del software

6.- El grupo deberá organizarse e investigar si existen muchas empresas que utilizan lenguajes no estructurados

7.- Escriba un ensayo con el tema: "Programadores y arte". Mencione que tanto arte se debe permitir en la labor ingenieril.

8.- Defina con sus propias palabras lo que es la programación estructurada

9.- Defina claramente la diferencia entre la programación modular y la programación estructurada

10.- Los programas que fueron escritos antes de la programación estructurada ¿Eran buenos? Escriba un ensayo sobre las caracteristicas de un buen programa.

a) Desde el punto de vista de una empresa

b) Desde el punto de vista de un usuario

c) Desde el punto de vista de un programador

11.- ¿Quién es considerado el padre de la programación estructurada?

12.- Si los resultados que entrega un programa son lo más importante para el usuario ¿Por qué es importante aplicar un enfoque disciplinado a un programa?

13.- ¿Por qué la eficiencia no es un factor de calidad de especial relevancia?

14.- ¿Por qué es necesario que un programa sea computable?

15.- Mencione un ejemplo de programa indecible

16.- ¿Cuál es la importancia de la correcta delimitación de los limites de un sistema?

17.- Explique por qué un programa es un modelo

18.- Explique por qué una computadora puede ser considerada una Máquina de Turing
19.- Explique por qué un computadora puede ser considerada un modelo de modelos
20.- ¿Qué impacto tiene la existencia de problemas indecibles en la computación?
21.- ¿Cómo explicaría a un gerente administrativo que una computadora solo puede resolver los problemas decibles?
22.- Realice una investigación sobre los distintos enfoques existentes para resolver programas. Elabore una lista con los nombres de los investigadores y el nombre de sus obras.
23.- Escriba un ensayo sobre el impacto que causa en los sistemas, el diseño orientado al procedimiento contra el diseño orientado a los datos
24.- Enuncie con sus propias palabras la diferencia entre producto de programación y programa
25.- ¿Qué es el análisis de requerimientos?
26.- ¿Cuáles son las consecuencias que existen cuando no se realiza una etapa de diseño previa?
27.- Explique ampliamente por qué los detalles de la implantación deben ser omitidos hasta el último momento.
28.- Explique ampliamente por qué es conveniente fragmentar los sistemas en módulos
29.- ¿Por qué es necesario un método de abstracción para encontrar la solución de un problema?
30.- ¿Por qué la elaboración de un sistema de información es una actividad compleja?
31.- Intente realizar más de una actividad a la vez, como ver un programa de tv, escuchar radio, platicar con otra persona, realizar un dibujo, balancear las piernas con un ritmo específico. ¿Cuántas actividades permite el cerebro antes de considerar que algo es complejo?¿Qué implicaciones tiene esto en el desarrollo de software?
32.- ¿Cuáles son las desventajas de los diagramas de flujo?
33.- ¿Cuáles son las ventajas del pseudocódigo?
34.- ¿Se deben "eliminar" también los diagramas estructurados?
35.- Mencione algunas prácticas que harían a un pseudocódigo ilegible
36.- Las matemáticas son un lenguaje preciso. ¿Por qué no se aboga por la escritura del pseudocódigo en un lenguaje matemático?
37.- ¿Por qué un enunciado de pseudocódigo debe consistir de un verbo y un sustantivo?
38.- Defina lo que es modularidad
39.- ¿Cómo se puede modularizar un problema?
40.- Explique la importancia del principio del diseño modular
41.- Defina lo que es un módulo
42.- ¿Por qué un módulo debe constar de un conjunto finito de entradas y salidas?
43.- Explique ampliamente la importancia de definir con precisión la función que debe cumplir un módulo
44.- ¿Por qué es necesario encontrar herramientas de diseño que sean independientes del lenguaje de programación.?
45.- ¿Por qué las relaciones entre los módulos aumentan el esfuerzo necesario para escribir un sistema?
46.- Explique la importancia de la independencia de los módulos
47.- Explique la importancia de la flexibilidad de los módulos
48.- ¿Cómo se hace un programa flexible?
49.- ¿Qué es acoplamiento?
50.- Explique ampliamente por qué la comunicación entre los módulos se debe realizar básicamente con listas de parámetros
51.- ¿Qué es cohesión?
52.- ¿En qué ocasiones la cohesión de un módulo puede atentar contra el acoplamiento en un sistema?
53.- Explique ampliamente por qué la cohesión de un módulo debe ser fuerte
54.- ¿Cuáles son los objetivos de la programación estructurada?
55.- ¿Por qué es importante el teorema de la estructura?
56.- Mencione una buena razón para violar el principio del programa propio
57.- Investigue en diversos lenguajes las estructuras de control disponibles. ¿Existe algún lenguaje en el cuál no existan las tres estructuras mencionadas en el teorema de la estructura?

58.- Piense usted en razones por las cuales algunos programadores veteranos violan indiscriminadamente el principio del programa propio

59.- ¿Cómo convencería usted a un programador que suele programar con saltos incondicionales, de las ventajas de la programación estructurada?

60.- Investigue que lenguajes admiten procesamiento paralelo y multitareas

61.- Si basta con tres estructuras para construir un programa. ¿Por qué considera usted que se expusieron cinco estructuras en el texto?

62.- Si basta con tres estructuras para construir un programa. ¿Por qué los manuales de los lenguajes de programación son tan grandes?

63.- Elija un producto de software y evalúe el acoplamiento, la cohesión y la modularidad de los sistemas

64.- Diséñese un experimento para evaluar cuánto tiempo ocupa un programa estructurado en las llamadas a los módulos que lo conforman

a) ¿Varía el tiempo cuando cambian los tipos de datos de los parámetros?

b) ¿Varía el tiempo entre llamadas secuenciales o recursivas?

65.- Suponga que está elaborando un manual sobre "estilo de programación". ¿Cuántos paréntesis permitiría en una expresión antes de que esta se volviera ilegible?

66.- Escriba un programa sin comentarios, que obtenga la matriz inversa y después pida a un amigo que lo lea y le diga qué es lo que hace dicho programa.

CAPITULO

II

LENGUAJE DE PROGRAMACION C

Si C tuviera escudo de armas
su lema podría ser
multum in parvo
Un mucho a partir de poco

Les Hancock

II.1 Elementos básicos de C

Dennis Ritchie inventó e implementó el primer compilador para lenguaje C en un DEC PDP-11 que usaba sistema operativo UNIX. La genealogía de C es fácil de trazar. Esta es su línea de ascendientes:

ALGOL 60

CPL
(Lenguaje de programación combinado)
Cambridge y la Universidad de Londres, 1963

BCPL
(Lenguaje básico de programación combinado)
Martin Richards, Cambridge, 1967

B (Ken Thompson)
Laboratorios Bell

C (Dennis Ritchie)
Laboratorios Bell

C es un lenguaje para programadores. Esta afirmación parece totalmente fuera de lugar ya que intuitivamente todos los lenguajes de programación deben ser para programadores. Sin embargo, al hacer un poco de ejercicio mental con ésta idea ya no se le encuentra tan disparatada. Sobre todo si pensamos en ciertos lenguajes que jamás fueron concebidos para los programadores. Un ejemplo claro de éste lo es COBOL. La enciclopedia de términos de microcomputación define a COBOL como: *un lenguaje narrativo en inglés basado en compilador [...]para la manipulación de datos y problemas de procesamiento.*(COBOL: COMMON BUSINESS ORIENTED LANGUAGE). Es decir, es una lenguaje que nació para teorio en la oficina de un ejecutivo de cualquier compania, con la finalidad de que dicha persona comprenda lo que el programa hace. Fué realizado para los hombres de negocios, no para los programadores.

Otro lenguaje de este tipo es BASIC (BEGGINNER'S ALL PURPOSE SYMBOLIC INSTRUCCION CODE). Un lenguaje que como su nombre lo indica, es un lenguaje para principiantes. BASIC se volvió popular precisamente por no ser un lenguaje complejo. Su extrema sencillez lo hace fácil de aprender. Pero en sus inicios no ofreció las herramientas mínimas para una programación disciplinada como la que se ha visto en el primer capitulo de estos apuntes.

C en cambio fué desarrollado en un laboratorio de investigaciones, por un programador que desarrolló un lenguaje pensando en los programadores. Tal vez C no sea estructurado tal y como se ha planteado con anterioridad, pero realmente es posible hacer un diseño modular, ya que está basado en el concepto de funciones.

En la misma referencia se define al lenguaje C de una manera un tanto paradójica:

Lenguaje C: *Lenguaje de programación de alto nivel que se parece al lenguaje ensamblador.*

Esta aparente contradicción proviene del hecho de que C es un lenguaje de alto nivel con características muy peculiares: es posible manipular los registros del microprocesador, puertos y, muchas otras características del hardware. En realidad, se considera a C como un lenguaje de nivel medio, ya que combina elementos de lenguajes de alto nivel con la funcionalidad del ensamblador.

Según hemos visto, C fué pensado para los programadores y no deja de sorprender que C no contiene operaciones para trabajar directamente con elementos compuestos, tales como cadenas de

caracteres, conjuntos, listas, arreglos o vectores, considerados como un todo. C no cuenta con operaciones de entrada-salida; no existen proposiciones READ y WRITE, ni métodos propios para el acceso a archivos. C sólo ofrece proposiciones (sentencias) de control de flujo sencillas, secuenciales, de selección, de iteración, bloques y subprogramas. Gracias a todo esto C es relativamente pequeño y puede aprenderse fácilmente.

Actualmente los paquetes más populares se realizan en lenguaje C. Programas como Dbase, Clipper, Lotus, etc, han sido manufacturados a través de é ste poderoso lenguaje que originalmente era utilizado para la programación de sistemas operativos (UNIX está desarrollado en C), intérpretes, editores, gestores de bases de datos, etc.

II.1.1 La función main()

El lenguaje C está basado en el concepto de funciones. Un programa C es una colección de una o más funciones. Cada función tiene un nombre y una lista de argumentos. En general se puede dar a una función el nombre que se quiera, con excepión de **main** (main: principal) , que se reserva para la función que inicia la ejecución del programa. Según se vió en el capítulo uno de estos apuntes, el nombre de cualquier función debe dar una idea aproximada de las tareas realizadas por esta.

La forma general de un programa C se muestra en la siguiente figura:

```
declaraciones globales
tipo-devuelto main(lista de parámetros)
declaraciones de parametros;
{
      secuencia de sentencias
}

tipo-devuelto nombre-de-función_1(lista de parámetros)
declaracion de parametros;
{
      secuencia de sentencias
}
.
.
.
tipo-devuelto nombre-de-función_n(lista de parámetros)
declaracion de parametros
{
      secuencia de sentencias
}
```

En dicha forma general se aprecia que salvo las declaraciónes globales, cada función presenta la misma estructura, incluyendo a la función `main()`.

La intención es comenzar a programar lo más pronto posible en lenguaje C y para eso comenzaremos con un programa muy sencillo.

Programa:
Escribir un programa que imprima las palabras "¡Hola amigos!"

Solución:

```
/*  programa saludo */
main()
{
      printf("\n ¡Hola amigos!");
}
```

Este programa presentará en la unidad de salida el mensaje `¡Hola amigos!` despues de haber saltado una línea.

Observaciones:

- `/* */` son los simbolos que engloban a los comentarios de un programa escrito en C. Estos simbolos son parte del estándar C.
- `main()` es el nombre de la función que en este momento no requiere de ningún tipo de parámetros de entrada. Tal y como se apuntó anteriormente, es necasario que exista en un programa la función main.
- `{ }` Las llaves delimitan el alcance de la función. Es decir. Todo lo que se encuentre las llaves pertenece a la función.
- `printf()` es una función llamada por `main()` a la cual se le pasa el parámetro `"\n ¡Hola amigos!"`
- La secuencia `\n` indica línea nueva porque `printf()` no ejecuta saltos de línea automáticamente.
- El punto y coma sirve para separar las sentencias en un programa C

Si por alguna causa el programa no funciona en su computadora puede intentar la siguiente variante

```
/*  programa saludo */
#include <stdio.h>
void main(void)
{
      printf("\n ¡Hola amigos!");
}
```

- La directiva `#include` indica que se hará uso de la lista de encabezados que se encuentra en el archivo descrito entre < >. El compilador requiere de estos encabezados para compilar correctamente las funciones del programa.
- `void` indica al compilador que la función no retomará un valor al terminar de ejecutarse y que tampoco requiere de un argumento de entrada

Hemos escrito el primer programa en lenguaje C. Recuerde que *aprender a programar es semejante a aprender a conducir, solo se logra frente a la máquina.*

II.1.2 Identificadores enteros y reales

El programa que hemos escrito presenta un severo inconveniente. Es demasiado rígido. Para lograr que tenga mayor flexibilidad requerimos de la asignación de valores que varien conforme se ejecuta ese programa. A estos valores variantes se les conoce como **variables** y los nombre con que se identifican como **identificadores de variable.**

El lenguaje C es sensible al tamaño. Esto significa que, en general, las mayúsculas y las minúsculas

son tomados como caracteres distintos. No es lo mismo para el compilador de C la variable **cambio** que **cambio.** En los compiladores orientados a PC y algunos multiusuario, permiten definir la sensibilidad al tamaño de tal forma que sea trascendente o no.

Un identificador de variable no puede ser igual a ninguna palabra reservada para el uso exclusivo del compilador. De tal forma que ninguna variable puede llamarse **main.** Esto podría crear confusión en el compilador al no saber como tratar al identificador en cuestión.

Los nombres de variable se forman empezando con una letra. No es permitido comenzar un identificador con un símbolo especial tal como >, <,), (, *, ;, etc. Es más dichos símbolos no son permitidos en ninguna parte del identificador. A continuación del primer caractér se puede escribir un dígito, un caractér de subrayado, o una letra. La longitud máxima depende del compilador. Es regla general suponer que la longitud mínima es de seis caracteres significativos en el nombre. Los nombres de variable pueden tener una longitud mayor a la máxima permitida pero el resto de los carateres no serán tomados en cuenta.

Supongamos que en un compilador que toma ocho caracteres se declaran dos variables como **contador1** y **contador2**. Entonces tendremos un error muy sutil y difícil de encontrar (a menos de que se disponga de un depurador). El programa compilará sin errores de sintaxis en cuanto a las dos variables se refiere, pero no funcionará correctamente. Esto es debido a que ambas variables son tomadas como lo mismo ya que sus primeros ocho caracteres significativos son iguales.

Todas las variables que se utilicen en un programa deben ser declaradas. Esto tiene como finalidad informar al compilador sobre el tipo de representación que tendrán los datos en forma interna y las operaciones válidas que les podrán ser aplicadas. Una declaración de tipo se puede realizar en tres partes distintas de un programa: dentro de las funciones, en la definición de parámetros de funciones y fuera de todas las funciones. En este momento veremos la primera de las formas de declaración dejando las restantes para el epígrafe II.3.3

Problema:
Defina cuales variables son válidas en C y cuales no lo son. Diga cuál es el problema existente en estas últimas.

nuevos$	C1fx24
2domenú	#empleado
cta_banco	conta.A
VaRIAbLe	numero_de_departamento

Solución:
correctas
C1fx24
cta_banco
VaRiAbLe
numero_de_departamento

incorrectas

nuevos$	contiene un caractér especial
2domenú	el primer caractér no es una letra
#empleado	el primer caractér es especial
conta.A	el punto tiene significado especial

Una declaración de tipo es la definición del tipo de datos que va a contener una variable. Dicha declaración tiene la forma general:

tipo: lista_de_variables

Donde

tipo es un tipo de datos válido en C
lista_de_variables consiste de uno o más identificadores separados por comas

En lenguaje C los enteros tienen el mismo significado que en matemáticas. Un entero no posee parte fraccionaria. En C no se posee una forma para representar el punto decimal en este tipo de números. La forma de declarar un entero es con la palabra reservada int (integer : entero). El rango que una variable entera puede abarcar depende de la máquina en la cual se esté trabajando. El estándar ANSI (ANSI: American National Standard Institute: Instituto Nacional Americano de Estándares) especifica también otros tipos de enteros. Estos se diferencian por el rango de valores que son capaces de soportar.

TIPO	TAMAÑO EN BITS	RANGO MINIMO
int	16	-32 767 a 32 767
unsigned int	16	0 a 65 535
signed int	16	igual que int
short int	16	igual que int
unsigned short int	8	0 a 65 535
signed short int	8	igual que short int
long int	32	-2 147 483 647 a 2 147 783 647
signed long int	32	-2 147 483 647 a 2 147 783 647
unsigned long int	32	0 a 4 294 967 295

Ejemplos de la declaración de variables se dan a continuación

```
int i,j,k;
unsigned int pago, cambio, billetes, monedas;
long int tot_ventas, subtotales;
```

Los identificadores reales soportan a números que poseen parte fraccionaria. La precisión de este tipo de datos suele darse en base al número de dígitos que es capaz de representar. Esta característica depende del tipo de representación interna en la máquina. Básicamente existen dos tipos de identificadores reales: los del tipo **float** (float: flotante. Hace referencia a la representación interna del punto decimal, que da la apariencia de estar flotando ya que no está representado físicamente) y **double**.(double: doble. Referencia a la doble longitud en la representación interna)

TIPO	TAMAÑO EN BITS	PRECISION
float	32	seis dígitos de precisión
double	64	diez dígitos de precisión
long double	128	diez dígitos de precisión

Una declaración para variables reales se realiza en base a la forma general vista anteriormente y no varía con respecto a la de los números enteros:

```
float centavos, porcentaje, pesaje;
double pi, area_circulo, raiz_real;
```

Una vez definida la manera como se pueden declarar las variables con las cuales se trabajará en el sistema debemos pasar al estudio de los operadores que manipularán dichas variables. C es un lenguaje muy rico en operadores. Sus características lo convierten en un lenguaje muy poderoso y que a la vez corre el riesgo de resultar en una codificación muy obscura. Debemos evitar caer en la tentación de escribir programas de este tipo (vease: LOS BUENOS PROGRAMAS).

El operador de asignación puede aparacer en cualquier expresión válida. Para realizar una asignación no es necesario dedicar una línea especialmente para el efecto. La asignación puede realizarse en forma anidada dentro de otras estructuras de control. La forma general de una asignación es:

variable = expresión;

Donde

variable es un nombre de variable o un puntero válido
expresión es una operación tan compleja como sea necesario o una simple constante

Las expresiones se forman con las variables y los operadores válidos para esas variables. Una expresión válida en lenguaje C pero no en matemáticas es `x=x+1`, que se lee *suma una unidad al valor actual de la variable x y entonces asigna ese valor a la variable x.*

En la tabla se muestran los operadores válidos para los números que hemos descrito hasta el momento

Operador	Acción
-	Resta binaria , también menos monario
+	Suma
*	Multiplicación
/	División
%	División en módulo
--	Decremento
++	incremento

Se puede apreciar que los cuatro operadores aritméticos comúnes: -, +, *, / se encuentran entre el conjunto de operadores de C. Estos se manejan de la misma manera que en cualquier otro lenguaje y tal como lo establece la aritmética. Los signos -, + y * se pueden aplicar indistintamente a las variables flotantes y a las enteras. Sin embargo una división con el operador (/) entre números enteros truncará la parte fraccionaria.

El operador % solo es posible aplicarlo a las variables enteras. Obtiene el residuo de una divisón. Sea por ejemplo el siguiente segmento de código.

```
x=7;
y=2;
z=x%y
cociente=x/y;
```

El resultado almacenado en `z` será uno en tanto que el almacenado en `cociente` será 3.

El incremento a uno de las variables suele escribirse como se mostró en parrafos anteriores:

```
x = x + 1;
```

C provee de un operador muy útil para esta tarea tan común. El operador ++ ejecuta la misma acción que la expresión anterior pero se escribe simplemente `x++` o `++x`.

Así mismo el operador `--` ejecuta la misma acción que

`x = x - 1` ⇨ `--x` ⇨ `x--`

En el uso de los operadores ++, -- dentro de expresiones es necesario precisar si van a preceder o a anteceder a la variable. Cuando el operador antecede a la variable, primero se realiza el incremento y después se utiliza dentro de la expresión. Cuando el operador precede a la variable primero se utiliza el valor de la variable dentro de la expresión y después se incrementa la variable. Supongamos que la variable **x** posee el valor de 10 antes de cada una de las siguintes expresiones:

```
w=x+10+x++;
y=++x+10+x;
z=(x++)+10+x
```

Después de estas asignaciones x=11, w = 30, y = 32, z = 30

Ejercicios:

Defina porque son incorrectas las siguientes expresiones:

```
main()
{
float x,y,z;

x=h*36.0;
y=12.0 % 3;
z = z + y / 0.0;
}
```

PRECEDENCIA

Se llama precedencia de una operación a la prioridad que posee dentro de la ejecución de una operación. Esta prioridad es resumida en la siguiente tabla:

TABLA DE PRECEDENCIA

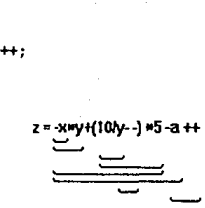
() Mayor precedencia
++ -- -(monario)
* / %
+ -
=

En general las expresiones se evalúan de izquierda a derecha pero no deben escribirse expresiones partiendo de dicha suposición ya que algunos compiladores optimizan código y esta regla puede dejar de cumplirse.

Problema:
Determinar el valor obtenido en la variable z en el siguiente segmento de código:

```
x=10;
y=3;
a=4;
z=-x*y+(10/y--)*5-a++;
```

Resultado: z = -19.



Problema:
Codificar la siguiente expresión:

$$x = 2 - \frac{x + 3}{bc + ad} - 15y$$

Solución:

```
x= 2*(  (-x+3)  /  (b*c  +  a*d) )  -  15*y
```

Observese que se han añadido espacios para facilitar la lectura de la expresión final. C acepta espacios y tabuladores a discreción con esa finalidad. Así mismo se pueden añadir paréntesis redundantes solo para asegurar la precedencia de evaluación de una subexpresión. Esto no afectará en nada el rendimiento de la máquina debido a que el compilador produce un código optimizado en donde los caracteres extras son eliminados.

Programa:
Escribir un programa que obtenga el área de un triángulo.

Solución:
Sabemos que el área de un triángulo se obtiene al evaluar la siguiente fórmula:

$$\text{area} = \frac{\text{base} * \text{altura}}{2}$$

```
/* Ejemplo 2 */
/* Ejemplo II.1.2 */
main()
{
      int base, altura;
      float area;
      base = 34;
      altura = 54;
      area = (base * altura) / 2.0;
      printf("\n Area de un triángulo");
      printf("\n Base: %d \n Altura: %d \n Area: %f";base, altura, area);
}
```

Observaciones:

Dentro del `printf()` se ha introducido **%d** para indicar que en ese lugar dentro del formato de salida se desea imprimir los dígitos enteros y **%f** para los números flotantes indicados en la lista de variables. Las variables enteras que constituyen la `base` y la `altura` entregarán decimales cuando se les divida sobre dos. Para que el programa sea capaz de contener dichos decimales se debe utilizar una variable real. La salida de este programa luce de la siguiente manera en la pantalla.

```
Area de un triángulo
Base: 34
Altura: 54
Area: 918.000000
```

II.1.3 Entrada y salida de datos

Como se había mencionado C no posee instrucciones para la entrada y la salida de los datos. Estas operaciones se realizan a través de funciones de biblioteca y proporcionan un amplio repertorio de opciones para el programador, permitiéndole escoger la opción que más se adapta al problema que desee solucionar.

En el primer ejemplo mostrado en este capítulo se presentó de una manera informal la función `printf()`, es momento de formalizarla un poco más.

SALIDA CON FORMATO

La forma general de la salida por formato es la siguiente.

print(cadena de control, lista de argumentos)

Donde:

cadena de control	Es una cadena con los códigos que controlarán la forma como de desplegarán los resultados en el dispositivo de salida
lista de argumentos	Es la lista con las variables o las expresiones que serán desplegadas

En la función `printf()` la cadena de control contiene los caracteres que se visualizan en la pantalla, las ordenes que le dicen a `printf()` cómo visulizar el resto de los argumentos o ambas cosas.

Problema:
Realizar un programa que realice la conversión de centímetros a pulgadas.

```
main()
/* ejemplo 3 */
{
      int centimetro;
      float  pulgada;
      centimetro = 15;
```

```
    pulgada = centimetro * 2.54;
    printf("\n Conversion de centimetros a pulgadas");
    printf("\n %d centimetros son %f pulgadas",centimetro,pulgada);

}
```

Resultados:

```
C:\COMP\TC2>ejem3

 Conversión de centimetros a pulgadas
 15 centimetros son 38.099998 pulgadas
```

Observaciones:

Tanto en este programa como en el inmediato anterior se tiene algo común en ciertos programas. La conversión de un tipo de dato en otro. La variable `centimetro` ha sido declarada como un número entero, al obtener su valor en `pulgada` el compilador ha realizado su conversión en forma automática para tener compatibilidad de tipos en la operación. En general C convierte todos los operandos al tipo del mayor operando. Ordenados de mayor a menor se tiene long double, double, float, unsigned long, long, unsigned.

A reserva de explicar más adelante el uso de cada especificador, se lista a continuación el conjunto de especificadores de formato.

Código	Formato
%c	Caractér
%d	Enteros decimales con signo
%i	Enteros decimales con signo
%e	Notación científica (e minúscula)
%E	Notación científica (E mayúscula)
%f	Coma flotante
%g	Usar %e o %f, el que resulte más corto
%G	Usar %E o %F, el que resulte 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	Mostrar puntero
%n	El argumento asociado es un puntero a entero al que se asigna el número de caracteres escritos
%%	Imprimir el signo %

Códigos de control

Es importante cuidar que en una lista de control una variable siempre tenga su correspondiente especificador de formato y que dicho formato sea compatible con el tipo de dato que al que se le va a dar

salida. Un entero definido como long no puede ser impreso solamente usando el especificador de campo **%d**. Es necesario anteponer el especificador **l** para indicar que se trata de un tipo largo. Este modificador de campo también puede anteponerse a **i,o,u** y **x**.

Programa:

Reescribir el programa de conversión de centímetros a pulgadas declarando los centímetros del tipo long int

Solución:

```
main()
/* ejemplo 4 */
{
      long int centimetro;
      float  pulgada;
      centimetro = 65915;
      pulgada = centimetro * 2.54;
      printf("\n Conversion de centimetros a pulgadas");
      printf("\n %ld centimetros son %f pulgadas",centimetro,pulgada);

}
```

Observaciones:

65915 es un número que no es soportado por el tipo int pero si por un tipo **long int**. Si se intenta utilizar esta cantidad con el tipo int no se desplegará un resultado correcto en la pantalla.

Obsérvese la **l** entre **%** y **d** en la especificación de campo.

La salida de este programa luciría aproximadamente así:

```
Conversión de centimetros a pulgadas
65915 centimetros son 167424.093750 pulgadas
```

ENTRADA CON FORMATO

La lectura desde un dispositivo externo se realiza a través de la función **scanf()**. Por su conducto es posible leer datos desde el teclado de un terminal. Y especificar ademas el tipo de dato que se desea recibir.

La forma general de la función **scanf()** es muy parecida a la de **printf()** y se muestra a continuación:

scan (cadena de contol, lista de argumentos)

Donde:

cadena de control	Es una cadena con los códigos que controlarán la forma como se recibirán los datos desde el dispositivo de entrada.
lista de argumentos	Es la lista con las direcciones de las variables que serán leídas.

La cadena de control le indica a **scanf()** que tipo de datos debe esperar para poder representarlos internamente de la manera correcta. La siguiente es una tabla de los especificadores de **scanf()**

Código	Formato
%c	Caractér
%d	Enteros decimales con signo
%i	Enteros decimales con signo
%e	Notación científica (e minúscula)
%f	Coma flotante
%g	Usar %e o %f, el que resulte más corto
%o	Octal sin signo
%s	Cadena de caracteres
%x	Hexadecimales sin signo (letras minúsculas)
%p	Mostrar puntero
%n	Recibe un valor entero igual al número de caracteres leídos
%u	Lee un entero sin signo
%[ ]	Muestra un conjunto de caracteres

Códigos de control para scanf()

Programa:
En una tienda departamental se desea instaurar un sistema de cómputo en el cual la cajera introduzca el total a pagar y la cantidad con la que paga el cliente, el programa debe dar por resultado el cambio que se le ha de entregar al cliente:

Solución:

```
/* ejemplo 6 */
/* Cálculo del cambio */
main()

{
      float total, pago, cambio;
      printf("\n Determinación del cambio \n Total a pagar : ");
      scanf("%f",&total);
      printf("\n Pago del cliente: ");
      scanf("%f", &pago);
      cambio= pago - total;
      printf("\n Cambio: %f ",cambio);
}
```

Resultados:

```
C:\COMP\TC2>ejem6
```

```
Determinación del cambio
Total a pagar : 123

Pago del cliente: 1000

Cambio: 877.000000
```

Observaciones:

Dentro de `scanf()` se coloca la cadena de control que, como hemos visto, tiene una forma similar a la de `printf()` . En la lista de variables se colocan la o las variables que han de ser leidos desde el teclado. Cada una de estas variables debe ser antecedida por un & para que `scanf()` asigne correctamente el valor en la localidad de memoria dedicada a esa variable. *Es muy importante este signo, ya que de no colocarlo podrían suceder cosas impredecibles.*

De la lista de especificadores podemos notar que un número entero puede ser leido con **%d** o con **%i**. Así mismo un número real puede ser leido con **%e, %f** o **%g.**

No siempre es desable tener tanta presición en el número de salida. Por ejemplo en la escritura de números que representen cantidades monetarias basta con dos decimales. Como casi todos los lenguajes, C permite definir una longitud de salida fija.

Ejemplo:
Defina la salida para el siguiente segmento de código

```
printf("\nEl número es: %3d", 1);
printf("\nEl número es: %1d",1);
printf("\nEl número es: %0d",1);
printf("\nEl número es: %10.3f",3.1416);
```

Solución:

```
El número es:   1
El número es: 1
El número es:         1
El número es:       3.146
```

Programa:
Escribir un programa que calcule el IVA a una cantidad de compras y entrege el total a pagar.

Solución:

```
/* ejemplo 5 */
/* Calcula total a pagar */
main()
{
     float subtotal, iva, total;
     printf("\n Cálculo de total de compra \n Subtotal : ");
     scanf("%f",&subtotal);
     iva=subtotal*0.1;
     total = subtotal + iva;
     printf("\n Iva: %7.2f \n Total a pagar: %7.2f", iva, total);
}
```

Resultados:

```
C:\COMP\TC2>ejem5

Cálculo de total de compra
Subtotal:1200

Iva  120.00
Total a pagar: 1320.00
```

II.1.4 Caracteres y cadenas

Frecuentemente es necesario disponer de variables que no contengan números sino letras o símbolos. A las variables que cotienen una sola letra, un solo dígito o un solo símbolo se les conoce como **variables caractér.**

La forma general para definir una variable caractér es la siguiente:

char lista_de_variables;

En el miembro derecho de un asignación a variable caractér puede ir una variable caractér, o un solo caractér encerrado entre apóstrofos.

Para hacer al lenguaje lo más independiente de la máquina en la cual se esté trabajando existe un código de barras invertidas. Aunque cada código se exprese como varios caracteres, estos en realidad son almacenados como un solo caractér. Estos códigos son mostrados a continuación:

Código	Significado
\b	Espacio atrás
\f	Salto de página
\n	Salto de línea
\r	Salto de carro
\t	Tabulador
\"	Comillas
\'	Apóstrofo
\0	Nulo
\\	Barra invertida
\v	Tabulador vertical
\a	Alerta (Sonido)
\N	Constante octal
\xN	Constante hexadecimal

Código de barras invertidas

La lectura de una variable caractér puede realizarse de muy distintas formas:

```
scanf("%c",&variable);
variable=getchar();
variable=getche();
variable=getch();
```

scanf(), getchar()

Estas dos funciones leen de la memoria del teclado (buffer) un caractér y lo almacenan en la variable. Ambas funciones esperan un retorno de carro para continuar. Estas dos formas de lectura de una variable caractér no son muy recomendables para entornos de programación interactiva. Estas funciones fueron escritas para el entorno de programación Unix que posee una filosofía para el manejo de las entradas y salidas basada en un concepto totalmente distinto al que utiliza el sistema DOS.

getch(), getche()

Estas dos funciones leen un solo caractér del teclado y son más recomendables para entornos interactivos. La función `getch()` no hace eco en la pantalla del caractér leído, es decir, no se visualiza en la pantalla lo que se digite en el teclado. La función `getche()` si visualiza el caractér en la pantalla.

Ejemplo:

```
/* Ejemplo 7 */
/* Uso de las variables caractér */
#include <stdio.h>
main()
{

      char nombre,a_paterno,a_materno,salto,signo,uno;

      int dos;
      salto='\n';
      signo='!';
      printf("\n Gritemos:");
      printf(" %c ¡ Hola mundo %c %c",salto,signo,salto);
      printf(" con alegria");
}
```

Resultado

```
Gritemos:
¡ Hola mundo !
con alegria
```

Ejemplo:

```
/* Ejemplo 8 */
/* Lectura de las variables caractér */
#include <stdio.h>
main()
{

      char letra;

      printf(" \n Pulse cualquier tecla para continuar");
      letra=getch();
      printf( "\n No se desplegó ningún caractér en pantalla");
```

```
      printf(" \n Ahora pulse cualquier caractér");
      letra=getche();
      printf(" \n El caractér leido fué:%c", letra);
}
```

Resultados:

```
C:\COMP\TC2>ejem8

 Pulse cualquier tecla para continuar <*>
 No se desplegó ningún caractér en pantalla
 Ahora pulse cualquier caractér*
 El caractér leido fué:*
```

En este ejemplo el programa pedirá que se pulse cualquier tecla sin desplegar el caractér en la pantalla. El siguiente caractér si será desplegado, tanto al momento de digitarse como al momento en que se alcance la función `printf()`.

Recibe el nombre de cadena de caracteres un conjunto de caracteres unidos como un solo objeto. A las variables que almacenan cadenas son conocidas como *arreglos de caracteres* o *variables string* (**string**: cadena). El apellido de una persona, su domicilio y su ocupación son ejemplos de cadenas de caracteres.

Lenguaje C almacena las cadenas en variables caractér reunidas bajo un mismo nombre. Lo único que diferencia a un caractér de otro es un subíndice. Cualquier cadena debe terminar con el caractér nulo. C no realiza validaciones sobre límites. Referido a las cadenas, si no se indica donde termina una cadena, el programa continuará vaciando información en el dispositivo de salida hasta que encuentre un caractér nulo.

Una variable de cadena debe declarase como caractér e indicando entre corchetes la longitud máxima que tendrá dicha cadena:

char nombre_de_cadena[longitud_do_cadena]

La lectura de una cadena se realiza por medio de **scanf()** y el indicador de campo **%s** o bien con la función **gets()**. La función **scanf()** unicamente leerá una cadena sin blancos. La función **gets()** leerá la cadena hasta alcanzar el retorno de carro.

Ejemplo:

```
/* Ejemplo 9 */
/* Lectura de las variables cadena */
#include <stdio.h>
main()
{

      char nombre[10];

      printf(" \n ¿Cual es su nombre? ");
      gets(nombre);
      printf( "\n Su nombre es: %s",nombre);

      printf(" \n Su inicial es: %c",nombre[0]);
}
```

Resultado:

```
C:\COMP\TC2>ejem9

¿Cual es su nombre? Ernesto

Su nombre es: Ernesto
Su inicial es: E
```

Observaciones:

En el programa anterior existe la variable de cadena **nombre** en donde se almacenarán 10 caracteres según lo indica la declaración. El conteo de los subindices inicia en 0 y llegará hasta el número 9. El último caractér de cualquier cadena en C es un nulo. El cual ha sido añadido en forma automática por la función **gets()** al final del último caractér leído.

Si se corre el programa y se le alimenta con el nombre Citlalli, los resultados serán:

```
¿Cuál es su nombre? Citlalli

Su nombre es: Citalli
Su inicial es: C
```

Los valores almacenados en las variables serán:

```
nombre[0]='C'
nombre[1]='i'
nombre[2]='t'
nombre[3]='l'
nombre[4]='a'
nombre[5]='l'
nombre[6]='l'
nombre[7]='i'
nombre[8]='\0'
nombre[9]=no definido
```

Ejemplo:

```
/* Ejemplo 10 */
/* Asignacion de las variables cadena */
#include <stdio.h>
main()
{

      /* Distintas formas de asignar una variable cadena */
        char nombre[10]="Ernesto";
      char a_materno[10];
      char a_paterno[10]={'P','e','ñ','a','l','o','z','a'};
```

```
        a_materno[0]='R';
        a_materno[1]='o';
        a_materno[2]='m';
        a_materno[3]='e';
        a_materno[4]='r';
        a_materno[5]='o';
        a_materno[6]='\0';


        printf( "\n Su nombre es: %s %s %s",nombre,a_paterno,a_materno);
        printf(" \n Sus iniciales son: %c. ",nombre[0]);
        printf("%c. %c. ",a_paterno[0],a_materno[0]);
}
```

Resultados:

```
C:\COMP\TC2>ejem10

 Su nombre es: Ernesto Peñaloza Romero
 Sus iniciales son: E. P. R.
```

Observaciones:

En este ejemplo se ha visto como es posible inicializar una variable en el mismo instante en que se declara. En el caso de la variable `nombre` la asignación se realiza a través de una constante de cadena. Dicha constante está encerrada entre comillas y no requiere de la indicación del caractér nulo ya que el compilador se la asigna automáticamente. Debe hacerse incapié en que el caractér nulo también debe estar contemplado al totalizar el largo máximo de la cadena, es decir, el caractér nulo también ocupa espacio. La asignación de la variable `a_paterno` se ha realizado dentro del bloque de programa. Esto se ha hecho asignando caractér por caractér a la variable que contiene a la cadena. Una forma altema de hacer exactamente lo mismo pero con una notación distinto es mostrado con la variable a_materno.

Algunas de las funciones más comunes para el manejo de cadenas son:

strcpy(cadena1,cadena2)	Copia cadena2 en cadena1	
strcat(cadena1,cadena2)	Concatena cadena2 al final de cadena1	
strlen(cadena)	Obtiene la longitud de la cadena	
strcmp(cadena1,cadena2)	Retoma:	igual a 0 si cadena1 = cadena
		menor que 0 si cadena1 < cadena2
		mayor que 0 si cadena1 > cadena2

Estas funciones se encuentran definidas en el archivo de cabecera `string.h`. Debe por tanto indicarse en una sentencia `#include` al inicio del archivo que las ocupe para que el compilador pueda encontrarlas. (#include <string.h>)

II.1.5 Constantes y comentarios

En el último ejemplo visto se visualizó que al momento de realizar una declaración de tipo también es posible inicializar una variable. En general esto es cierto para todos los tipos de datos. La forma general de una inicialización es:

tipo nombre_de_variable = constante;

Aquellas variables que no sean inicializadas tendrán un valor indefinido antes de su primera asignación (aunque esta ocurra hasta la línea 1265).

Una constante es un valor que no cambia a lo largo de la ejecución de un programa. Cuando un valor es definido como una constante no será posible realizar modificaciones sobre el mismo durante la ejecución del programa. Por ejemplo, pódemos definir que `PI = 3.1415` o que `IVA = 0.10`. estos son valores que no debieran cambiar y que es deseable definir en un punto específico del programa. Un programador que utilice frecuentemente una cantidad constante, como el IVA, necesita definirlo una sola vez y despues usar su nombre simbólico. Si por alguna razón este valor cambia (por motivos fiscales o de precisión) se tendrá que modificar en una sola línea y no en todos los lugares en donde se haga referencia a dicho valor.

El estándar definido por Kernigal & Richie no incluye la palabra clave reservada **const** con la cual se define a una constante. El estándar ANSI sí la especifica. Lea su manual de referencia para saber si está incluida o no.

Una constante se define a través de la palabra reservada **const** de la siguiente forma general

const tipo nombre_de_variable = constante;

Si su implantación no cuenta con este modificador de tipo, declare la variable inicializandola en la declaración de tipo. Esto no le asegurá que la variable (y la llamamos variable) no será modificada.

En general un comentario se expresa encerrandolo entre `/* */`. Entre la barra y el asterisco no puede ir un espacio en blanco. Un comentario puede ir en cualquier lugar del programa excepto en medio de una palabra reservada.

En las implantaciones que utilizan C++ existe el comentario de línea, el cual solo requiere del inidicador de inicio y que puede ir en la misma línea de una instrucción.

Ejemplo:

```
/* Ejemplo 11 */
/* Uso de las constantes y los comentarios */
#include <stdio.h>
main()
{

      /* Distintas formas de asignar una variable cadena */
      const float iva = 0.1;  // Declaracion valida solo en C ANSI
      float precio,total;
```

```
    /* En caso de que el comentario no funcione
    se debe sustituir por el comentario estandar */

    printf("\n Caja\n Subtotal: ?");
    scanf("%f",&precio);
    total = precio*(1+iva);      //Comentario válido sólo en C++
    printf( "\n El importe total es : %10.2f",total);
}
```

Preguntas:

1.- ¿Quién escribió el primer compilador de lenguaje C?
2.- ¿Porqué se dice que el lenguaje C es para programadores?
3.- ¿Porqué es necesaria la declaración tipo cuando se define una variable?
4.- ¿ Porque es necesario definir el tipo de datos a emplear?
5.- ¿ Porqué existen tantos tipos de enteros y de números reales?
6.- Investige la representación interna de los números en el compilador que utilice.
7.- ¿Es conveniente mezclar en las expresiones matemáticas distintos tipos de datos?
8.- Explique porqué no es recomendable es escribir sentencias como la siguiente:

```
printf("\n El valor es: %d", x = y++);
```

9.-¿ Porqué es importante la precedencia de las operaciones?
10.- ¿Cuál es la precedencia de los operadores vistos en esta sección?
11.- ¿Qué sucede si una cadena no termina con un caracter nulo?

II.2 Estructuras de control de programa.

En esta sección se estudiarán las estructuras de control de un programa. Este apartado es la aplicación práctica de los conceptos que se han discutido en el capítulo uno.

II.2.1 La estructura FOR

La sentencia que se estudiará en primer término corresponde a la iteración FOR la cual se ejecutará cero o más veces dependiendo de una expresión. La sintaxis es la siguiente:

for (expresion1; expresion2;expresion3) sentencia;

Donde:

expresion1 : es la expresión evaluada al inicio del ciclo
expresion2 : es la expresion que detendrá al ciclo cuando esta evalúa en cero
expresión3 : es una expresion que será evaluada cada vez que se ejecute un nuevo ciclo.
sentencia : es la sentencia que conforma el cuerpo del for y que se desea ejecutar.

En lenguaje C se dice que una variable es falsa cuando esta tiene el valor de cero. Una variable será verdadera cuando tenga un valor distinto de cero. En general cuando se evalúa una expresión cero corresponde a falso.

Programa:

Reescribir el programa de conversión de centímetros a pulgadas para que calcule una tabla de 20 números con incrementos de 5 centímetros.

Solución:

```
/* ejemplo 12 */
/* Tabla de conversión */
main()

{
int centimetro;
printf("\n\tConversión de centimetros a pulgadas");
printf("\n\t\t Tabla de conversiones");
centimetro=1;
printf("\n\t %3d centimetros son %7.2f pulgadas",centimetro,centimetro*2.54);
for (centimetro=5; centimetro <= 100; centimetro=centimetro+5)
  printf("\n\t %3d centimetros son %7.2f pulgadas",centimetro,centimetro*2.54);
}
```

Resultados:

```
C:\COMP\TC2> ejem12
        Conversión de centimetros a pulgadas
              Tabla de conversiones
            1 centimetros son    2.54 pulgadas
            5 centimetros son   12.70 pulgadas
           10 centimetros son   25.40 pulgadas
           15 centimetros son   38.10 pulgadas
           20 centimetros son   50.80 pulgadas
           25 centimetros son   63.50 pulgadas
           30 centimetros son   76.20 pulgadas
           35 centimetros son   88.90 pulgadas
           40 centimetros son  101.60 pulgadas
           45 centimetros son  114.30 pulgadas
           50 centimetros son  127.00 pulgadas
           55 centimetros son  139.70 pulgadas
           60 centimetros son  152.40 pulgadas
           65 centimetros son  165.10 pulgadas
           70 centimetros son  177.80 pulgadas
           75 centimetros son  190.50 pulgadas
           80 centimetros son  203.20 pulgadas
           85 centimetros son  215.90 pulgadas
           90 centimetros son  228.60 pulgadas
           95 centimetros son  241.30 pulgadas
          100 centimetros son  254.00 pulgadas
```

Observaciones:

El programa define únicamente una variable `centimetro` con la cual se realizan todas las operaciones. Dentro del `printf()` encontramos la expresión que realiza la conversión aunque esta no es asignada a ninguna variable. Dentro del for se tiene que la primera expresión (de inicialización de variables) se asigna el valor de 5 a `centimetro;` el bucle se realizará mientras esta variable sea menor o igual que 100. Al término de cada ciclo la variable se incrementará en cinco unidades.

Observese que se incluyen expresiónes de asignación dentro de la declaración de control del ciclo for. Es más, esta sentencia permite diversas variantes como podría ser el incluir más de una expresión en cualquiera de los tres bloques; para hacerlo solo se debe separar cada sentencia por una coma.

El ciclo for no necesariamente debe contener una sola sentencia en su bloque de ejecución. Para incluir más de una sentencia se hace uso de los indicadores de bloque. La llave derecha ({) indica el inicio de un bloque de sentencias, su contraparte lo es la llave izquierda (}).

Programa:

Se desea hacer un programa que evalúe el cuadrado de los números que van de -20 hasta 20 con incrementos unitarios. El resultado se desplegará en una tabla que muestre que la función es simétrica , es decir que $x^2 = (-x)^2$.

```
/* ejemplo 13 */
/* Tabla de x cuadrada */
main()

{
      int num_pos, num_neg, r_pos, r_neg;
      printf("\n\t\t   Tabla comparativa \n\t   x \t x^2 \t\t   x \t x^2");

      /*  Ciclo del programa     */
      for (num_pos=1,num_neg=-1 ;num_pos <=20 ;num_pos++,num_neg-- )
      {
            /* se evalua el cuadrado de los numeros
                positivos y los negativos              */
            r_pos=num_pos*num_pos;
            r_neg=num_neg*num_neg;

            printf("\n\t %3d \t %3d",num_pos,r_pos);
            printf("\t\t %3d \t %3d",num_neg,r_neg);
      }
      printf( "\n\n Fin de tabla comparativa");
}
```

Resultados:

```
          Tabla comparativa
   x     x^2          x     x^2
   1       1         -1       1
   2       4         -2       4
   3       9         -3       9
   4      16         -4      16
   5      25         -5      25
   6      36         -6      36
   7      49         -7      49
   8      64         -8      64
   9      81         -9      81
  10     100        -10     100
  11     121        -11     121
  12     144        -12     144
  13     169        -13     169
  14     196        -14     196
  15     225        -15     225
  16     256        -16     256
  17     289        -17     289
  18     324        -18     324
  19     361        -19     361
  20     400        -20     400

Fin de tabla comparativa
```

Observaciones:

Las tres partes de la declaración de control del ciclo FOR aún existen, cada una está separada por el punto y coma. En la sección de inicialización se asigna 1 a `num_pos` y -1 a `num_neg`. La condición de salida es que `num_pos` debe ser menor o igual a 20 . El la tercera parte se tiene que en cada ciclo de ejecución `num_pos` se incrementa en una unidad en tanto que `num_neg` se decrementa en una unidad.

Debe tenerse cuidado con este tipo de facilidades que ofrece el lenguaje ya que puede crear programas demasiado difíciles de leer. En la mayoría de los casos es preferible que el programa tarde 10 ms más en ejecutar, a que el programador que le dé mantenimiento tarde 30 minutos más en entenderlo.

El mismo programa podría reescribirse de la siguiente manera:

```
/* ejemplo 14 */
/* Tabla de x cuadrada. Version 2 */
main()

{
      int num_pos, num_neg, r_pos, r_neg;
      printf("\n\t\t   Tabla comparativa \n\t    x \t x^2 \t\t   x \t x^2");
```

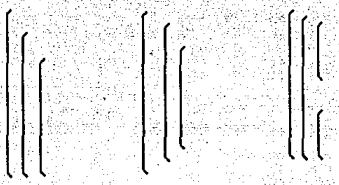
```
    /*  Ciclo del programa     */
    num_neg = 0;
    num_pos =  0;
    for ( ;num_pos <=20 ; )
    {
          /* se evalua el cuadrado de los números
             positivos y los negativos      */
          num_pos++;
          num_neg--;
          r_pos=num_pos*num_pos;
          r_neg=num_neg*num_neg;

          printf("\n\t %3d \t %3d",num_pos,r_pos);
          printf("\t %3d \t %3d",num_neg,r_neg);
    }
    printf( "\n\n Fin de tabla comparativa");
}
```

Observaciones:

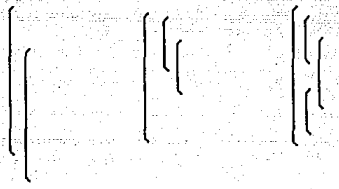
Ahora el programa no tiene nada en dos de las expresiones de control. Esto es posible porque las variables han sido manipuladas fuera del ciclo. Una vez más se hace una alerta para evitar este tipo de codificación. No es solo por el hecho de que el ciclo no cumple con la forma canónica con la que se está acostumbrado a ver a un ciclo for, sino porque la codificación se toma bastante obscura y poco portátil. Cuando un programador usa un ciclo for, suele tener en mente que se trata de un bloque de instrucciones que será ejecutado un número finito y predictible de veces. Este número de repeticiones vienen claramente establecidas dentro de los límites del for. Si un programador no observa dichos límites en *su lugar* tendrá que buscar en otras partes del código cuales son las condiciones iniciales y la instrucción de incremento de la variable de control.

Dentro de un bloque de instrucciones puede colocarse cualquier sentencia ejecutable. Esto incluye una sentencia FOR. A esto se le llama anidamiento de sentencias. En un caso así se ejecutará primero la sentencia más interna y posteriormente la más externa.

Existen restricciones importantes en cuanto al alcance de un ciclo FOR cuando se encuentra anidado. El anidamiento de un ciclo for esta delimitado por el principio de la modularidad y el programa propio, es decir, un ciclo no puede rebasar los límites de control de un ciclo más externo.



Anidamientos válidos



Anidamientos no válidos

Programa:
Hacer un programa que escriba las diez tablas de multiplicar.

Pseudocódigo:

```
Tablas de multiplicar

DECLARA VARIABLES factor_1, factor_2, producto, espera

DESPLIEGA(" Tablas de multiplicar")
HAZ PARA (CADA UNA DE LAS TABLAS)
      DESPLIEGA(TABLA DE MULTIPLICAR DEL factor_1);
      HAZ PARA (CADA NÚMERO DE LA TABLA)
            EVALUA EL producto
            DESPLIEGA(producto)
      FIN DEL HAZ

      ESPERA TECLA PARA CONTINUAR
FIN DEL HAZ
DESPLIEGA( " Fin de tablas de multiplicar")
}
```

Solución:

```
/* ejemplo 15 */
/* Tablas de multiplicar */
main()

{
int factor_1, factor_2, producto;
char espera;

printf("\n\t\t Tablas de multiplicar");

/*  En este ciclo se generan una por una las tablas  */
for (factor_1=1 ; factor_1 <= 9 ; factor_1++ )
{
      printf("\n\t\t Tabla del %2d",factor_1);

      /* En este ciclo se evalua cada producto de la tabla*/
      for (factor_2=1 ; factor_2<=10 ; factor_2++)
      {
            producto=factor_1*factor_2;
            printf("\n\t\t %3d  x %3d =%3d",factor_1,factor_2,producto);
      }

      printf("\n\n Pulse una tecla para continuar...");
      espera=getch();

}
printf( "\n\n Fin de tablas de multiplicar");
}
```

Resultados:

```
        Tablas de multiplicar
        Tabla del  1
          1  x   1 =  1
          1  x   2 =  2
          1  x   3 =  3
          1  x   4 =  4
          1  x   5 =  5
          1  x   6 =  6
          1  x   7 =  7
          1  x   8 =  8
          1  x   9 =  9
          1  x  10 = 10

 Pulse una tecla para continuar...
  .
  .
  .

 Pulse una tecla para continuar...
```

```
Tabla del  9
  9  x   1 =  9
  9  x   2 = 18
  9  x   3 = 27
  9  x   4 = 36
  9  x   5 = 45
  9  x   6 = 54
  9  x   7 = 63
  9  x   8 = 72
  9  x   9 = 81
  9  x  10 = 90

Pulse una tecla para continuar...

Fin de tablas de multiplicar
```

Observaciones:

Dado que el programa imprime las nueve tablas de multiplicar, por simplicidad se muestran solo los resultados extremos.

El programa imprime el titulo **Tablas de multiplicar** y entra al ciclo FOR inicializando la variable `factor_1` a uno. Al encontrar la llave derecha el compilador sabe que debe ejecutar todo el bloque de instrucciones. Imprime el letrero `Tabla del` y sustituye el valor que posea la `factor_1` en el especificador de campo. Para la primer pasada será 1. Encuentra un segundo ciclo que inicializa `factor_2` en uno y entra al segundo ciclo. Este nuevo ciclo FOR no regresará el control al FOR más externo en tanto no se cumpla que `factor_2` sea mayor a 10.

Al entrar a su bloque de asignaciones realiza el producto de `factor_1` y `factor_2`. En este caso inicial ambos tienen asignado el valor de 1 y por tanto el producto es también la unidad. Se despliega en la unidad de salida este resultado y regresa al bloque de control del for más interno. Allí incrementa `factor_2` en una unidad y vuelve a evaluar el producto de `factor_1` y `factor_2`.

Este ciclo continua hasta que se terminan de evaluar los diez productos de la tabla del uno. al finalizar, se pide que se oprima una tecla y el ciclo más externo vuelve a empezar todo de nuevo. Asigna `factor_1` igual a 2 y se procede al cálculo de la tabla del 2 y así sucesivamente hasta que `factor_1` es igual a nueve y se calcula la tabla del nueve

Un ciclo puede inclusive no entrar jamás a su cuerpo de instrucciones. Observese la siguiente variación al mismo programa:

Pseudocódigo:

```
Tablas de multiplicar

DECLARA VARIABLES factor_1, factor_2, producto, espera

DESPLIEGA(" Tablas de multiplicar")
LEE (tabla_inicial, tabla_final)
```

```
HAZ DESDE (tabla_inicial HASTA tabla_final)
      DESPLIEGA(TABLA DE MULTIPLICAR DEL factor_1);
      HAZ PARA (CADA NÚMERO DE LA TABLA)
            EVALUA EL producto
            DESPLIEGA(producto)
      FIN DEL HAZ

      ESPERA TECLA PARA CONTINUAR
FIN DEL HAZ
DESPLIEGA( " Fin de tablas de multiplicar")
}



/* ejemplo 16 */
/* Tablas de multiplicar.Variación */
main()

{
int factor_1, factor_2, producto;
int t_inicial,t_final;
char espera;

printf("\n\t\t Tablas de multiplicar");

printf("\nSe imprimirán las tablas de multiplicar");
printf("\n  Tabla de inicial: ");
scanf("%d",&t_inicial);
printf("\n  Tabla de final  : ");
scanf("%d",&t_final);

/*  En este ciclo se generan una por una las tablas  */
for (factor_1=t_inicial ; factor_1 <= t_final ; factor_1++ )
{
      printf("\n\t\t Tabla del %2d",factor_1);

      /* En este ciclo se evalua cada producto de la tabla*/
      for (factor_2=1 ; factor_2<=10 ; factor_2++)
      {
            producto=factor_1*factor_2;
            printf("\n\t\t %3d  x %3d =%3d",factor_1,factor_2,producto);
      }

      printf("\n\n Pulse una tecla para continuar...");
      espera=getch();

}
printf( "\n\n Fin de tablas de multiplicar");
}
```

Resultados:

```
C:\COMP\TC2>ejem16

                Tablas de multiplicar
Se imprimir n las tablas de multiplicar
  Tabla de inicial: 5

  Tabla de final  : 7

                Tabla del  5
                  5  x   1 =  5
                  5  x   2 = 10
                  5  x   3 = 15
                  5  x   4 = 20
                  5  x   5 = 25
                  5  x   6 = 30
                  5  x   7 = 35
                  5  x   8 = 40
                  5  x   9 = 45
                  5  x  10 = 50

 Pulse una tecla para continuar...
                Tabla del  6
                  6  x   1 =  6
                  6  x   2 = 12
                  6  x   3 = 18
                  6  x   4 = 24
                  6  x   5 = 30
                  6  x   6 = 36
                  6  x   7 = 42
                  6  x   8 = 48
                  6  x   9 = 54
                  6  x  10 = 60

 Pulse una tecla para continuar...
                Tabla del  7
                  7  x   1 =  7
                  7  x   2 = 14
                  7  x   3 = 21
                  7  x   4 = 28
                  7  x   5 = 35
                  7  x   6 = 42
                  7  x   7 = 49
                  7  x   8 = 56
                  7  x   9 = 63
                  7  x  10 = 70

 Pulse una tecla para continuar...

 Fin de tablas de multiplicar
```

Observaciones:

En este caso si el usuario digita como valor inicial a `t_inicial` el número 6 y 2 a `t_final` el programa no imprimirá ninguna tabla de multiplicar

El uso más recomendable para una estructura for es para la ejecución de bloques cuyo número de iteraciones sea conocido de antemano. Aunque este caso, tal y como se vió en el capítulo uno solo sea un caso particular de un bucle. No obstante existen en la práctica muchos casos en los cuales la sentencia for finita y determinada no solo es deseable sino necesaria.

II.2.2 Arreglos

Hemos visto ya lo que es un arreglo y su manejo aunque de una manera informal. El manejo de las cadenas de caracteres se realiza a través de arreglos. Un arreglo en una colección de variables del mismo tipo, con el mismo nombre, y que se diferencian unos de otros a través de un subíndice.

La forma general para declarar un arreglo es:

tipo nombre_de_variable[número_de_elementos]

Cuando el compilador encuentra una declaración de tipo para un arreglo reserva una cantidad de espacio suficiente en la memoria para poder contenerlo. En general C asigna al primer elemento de un arreglo la localidad de memoria más baja y al último elemento la localidad más alta. Todo el arreglo es asignado en localidades contiguas de memoria.

Contra lo que sucede en otros lenguajes, el primer elemento de un arreglo en C no tiene al subíndice 1 sino al subíndice 0. Esto implica que si se declara un arreglo con 10 elementos el último subíndice válido que se puede utilizar es el 9. C nos permite accesar cualquier elemento fuera de los límites del arreglo, pero los datos no son válidos. C no valida los límites de sus arreglos; dicha validación es responsabilidad del programador. Este aspecto es muy importante cuando se esté manejando información dentro de la memoria ya que si por falta de una validación adecuada un arreglo es traspasado, se podría estar escribiendo en un dato útil para algún segmento de programa. Esto acarrearía a la larga un error en los datos. En el peor de los casos se podría sobreescribir en el código ejecutable y el programa terminará produciendo resultados inesperados o inclusive provacoando una caída del sistema.

Programa:
En cierta coorporación financiera se requiere obtener las ventas acumuladas por mes. Los directivos desean ver en un tabla dos columnas. La primera de ellas reflejará las ventas generadas en el mes y en la segunda las cantidades que se acumulen a lo largo del año.

Pseudocódigo:

```
Tabla de acumulados
DECLARA VARIABLES
DESPLIEGA (" Acumulados mensuales")
HAZ PARA (TODOS LOS MESES)
      LEE( cantidad (mes) )
FIN DEL HAZ
DESPLIEGA encabezado DE LA TABLA
HAZ PARA ( TODOS LOS MESES)
      CALCULA acumulado
      DESPLIEGA acumulado
FIN DEL HAZ
```

Solución:

```
/*ejemplo 17 */
/* Una tabla de acumulados */
main()
{
int cantidad[12],  contador;
long int suma;
printf("\n\t\t Acumulados mensuales");

for (contador=0; contador <= 11 ; contador++)
{
      printf("\n Digite las ventas del mes %2d :",contador+1);
      scanf("%d",&cantidad[contador]);
}
suma=0;
printf("\n\n\t\t TABLA DE ACUMULADOS");
printf("   \n\t\t ====================");
printf("\n\n \t Mes          Ventas    Acumulados");
for (contador=0; contador <= 11 ; contador++)
{
      suma=suma+cantidad[contador];
      printf("\n\t [%2d]   %10d %10d",contador+1,cantidad[contador],suma);

}
}
```

Resultados:

```
                Acumulados mensuales
 Digite las ventas del mes  1 :1200

 Digite las ventas del mes  2 :1000

 Digite las ventas del mes  3 :1320

 Digite las ventas del mes  4 :1450

 Digite las ventas del mes  5 :1250

 Digite las ventas del mes  6 :1400

 Digite las ventas del mes  7 :1600

 Digite las ventas del mes  8 :2000

 Digite las ventas del mes  9 :2150

 Digite las ventas del mes 10 :1000

 Digite las ventas del mes 11 :1500

 Digite las ventas del mes 12 :2400
```

```
                TABLA DE ACUMULADOS
                ===================

     Mes          Ventas     Acumulados
     [ 1]         1200         1200
     [ 2]         1000         2200
     [ 3]         1320         3520
     [ 4]         1450         4970
     [ 5]         1250         6220
     [ 6]         1400         7620
     [ 7]         1600         9220
     [ 8]         2000        11220
     [ 9]         2150        13370
     [10]         1000        14370
     [11]         1500        15870
     [12]         2400        18270
```

Un arreglo puede tener más de una dimensión, de tal manera que forme matrices de dos, tres o aún más dimensiones. Generalmente los arreglos no suelen recomendarse para más de tres dimensiones ya que se vuelven muy difíciles de entender, aún para el programador que los declaró.

La forma general para declarar un arreglo multidimensional es:

tipo nombre_del_arreglo[d1][d2][d3][d4]...[di]...[dn]
Donde
d es la longitud del arreglo en la i-ésima dimensión
n es la cantidad de dimensiones que contiene el arreglo

Programa:
Hacer un programa que efectue la suma de dos matrices bidimensionales.

Pseudocódigo:

```
SUMA MATRICES
DECLARA  matriz1, matriz2,suma,renglon, columna,orden1,orden2
ESCRIBE TITULAR(" Suma de dos matrices")
LEE (orden1 ,orden2)
HAZ PARA (cada renglon de la matriz1)
      HAZ PARA CADA (columna de la matriz1)
            LEE(matriz1[renglon][columna])
      FIN DEL HAZ
FIN DEL HAZ

HAZ PARA (cada renglon de la matriz2)
      HAZ PARA CADA (columna de la matriz2)
            LEE(matriz2[renglon][columna])
      FIN DEL HAZ
FIN DEL HAZ
```

```
HAZ PARA CADA ( RENGLON DE LA matriz1)
      HAZ PARA CADA ( COLUMNA DE LA matriz2)
            SUMA LOS ELEMENTOS
      FIN DEL HAZ
FIN DEL HAZ
ESCRIBE (matriz1)
ESCRIBE (matriz2)
ESCRIBE (matriz_resultado)
```

Solución:

```
/*Ejemplo 18 */
/* Suma de matrices */
/* 6/nov/1992 */
#include <stdio.h>
main()
{
int matriz1[5][5], matriz2[5][5],suma[5][5];
int renglon, columna,orden1,orden2;

printf("\n\t\t Suma de dos matrices");

/* Las dos matrices deben ser conformables */
printf("\nDigite el orden de la matriz [2-5] x [2-5] ");
scanf("%d",&orden1);
printf("%d x ",orden1);
scanf("%d",&orden2);

/* En C los índices comienzan con 0 por lo tanto se deben decrementar el orden */
orden1--;
orden2--;

/* Captura de cada elemento de la matriz A */
printf("\n\n Captura de la matriz A");

for (renglon=0; renglon <= orden1 ; renglon++)
    for (columna=0; columna <=orden2; columna++)
      {
      printf("\n Digite el elemento (%2d,%2d) :",renglon+1,columna+1);
      scanf("%d",&matriz1[renglon][columna]);
      }


/* Captura de los elementos de la matriz B */
printf("\n\n Captura de la matriz B");

for (renglon=0; renglon <= orden1 ; renglon++)
    for (columna=0; columna <=orden2; columna++)
      {
      printf("\n Digite el elemento (%2d,%2d) :",renglon+1,columna+1);
      scanf("%d",&matriz2[renglon][columna]);
      }
```

```
printf(" \n RESULTADO:\n\n\n");

/* Suma de los elementos */
for (renglon=0; renglon <= orden1 ; renglon++)
    for (columna=0; columna <=orden2; columna++)
      {
      suma[renglon][columna]=matriz1[renglon][columna]+matriz2[renglon][columna];
      }

/* Despliege de la matriz A */
printf("\nMatriz A:\n");

for (renglon=0; renglon <= orden1 ; renglon++)
    {
    for (columna=0; columna <= orden2; columna++)
      printf(" %10d ",matriz1[renglon][columna]);
    printf("\n");
    }

/* Despliege de la matriz B */
printf("\nMatriz B:\n");
for (renglon=0; renglon <= orden1 ; renglon++)
    {
    for (columna=0; columna <= orden2; columna++)
      printf(" %10d ",matriz2[renglon][columna]);
    printf("\n");
    }

/* Despliege de la matriz resultado */
printf("\nMatriz Suma:\n");
for (renglon=0; renglon <= orden1 ; renglon++)
    {
    for (columna=0; columna <= orden2; columna++)
      printf(" %10d ",suma[renglon][columna]);
    printf("\n");

    }
}
```

Resultados:

```
C:\>EJEM18

                Suma de dos matrices
Digite el orden de la matriz [2-5] x [2-5] 2
2 x 2
```

```
Captura de la matriz A
Digite el elemento ( 1, 1) :10

Digite el elemento ( 1, 2) :12

Digite el elemento ( 2, 1) :13

Digite el elemento ( 2, 2) :15


Captura de la matriz B
Digite el elemento ( 1, 1) :20

Digite el elemento ( 1, 2) :23

Digite el elemento ( 2, 1) :22

Digite el elemento ( 2, 2) :21

 RESULTADO:



Matriz A:
        10          12
        13          15

Matriz B:
        20          23
        22          21

Matriz Suma:
        30          35
        35          36
```

Observaciones:

En una matriz multidimensional es necesario manipular cada uno de los índices en forma separada. En otras palabras, es necesario que en lecturas como la que este programa requiere, se aniden dos estructuras de repetición para poder manejar cada índice. No es trascendente si primero se leen las columnas o primero los renglones. Para el compilador no es trascendente, pero para el humano que requiere realizar esta captura si es importante ya que al escoger una convención en lugar de otra puede acarrear problemas con los papeles que el capturista usualmente tiene en su mesa de trabajo. Generalmente se captura por renglones como se ha mostrado en el programa.

También es muy importante que se respeten las convenciones de captura. Si en una parte del programa se piden las matrices por renglón, no deben existir módulos en los cuales la captura o el despliege de la información se realice en forma de columnas. Esta situación crea una verdadera confusión en el usuario. y permite la introducción de errores que bien podrían ser evitados.

II.2.3 Expresiones de relación y lógicas.

Hasta este momento se han manejado de una manera informal las expresiones de relación y lógicas. Para definirlas de una manera formal diremos que:

Una *operación de relación* es aquella que manipula la información de valor de los operandos. En los ciclos FOR , la expresión de salida es una *condición* que debe ser satisfecha para que el ciclo puede ser vuelto a ejecutar. El valor que puede adquirir cualquier expresión de relación puede ser verdadera o falsa.

Los operadores relacionales son muy sencillos de aprender debido a los cercano que se encuentran de la vida cotidiana. La siguiente tabla muestra una lista de dichos operadores:

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

Tabla de operadores

Entre las cosas que más llaman la atención al observar esta tabla se puede notar que la igualdad de relación debe planterse con un doble signo de igualdad. Recordemos que el signo de asignación no requiere de un lugar específico para su uso. Sin el uso del doble signo de igualdad el compilador no podría establecer la diferencia entre una asignación y una pregunta de relación.

Ejemplos:

total > 100000 letra == 'a'

Las dos siguientes expresiones son equivalentes.:

(iva+isapt) !=max_gravable iva+ispt != max_gravable

En los ejemplos vistos el resultado solo puede tener dos valores: cierto o falso. Esto es bien independiente del tipo de dato que se este manipulando. Estos dos resultados reciben el nombre de *Resultados lógicos ó Booleanos.* En C un valor de falso es representado con un cero y un valor equivalente a cierto es representado con cualquier elemento distinto a cero.

Existen tres operadores que manipulan exclusivamente esos valores de verdad. Los operadores lógicos son: !, ||, && (NO, O, Y).Los cuales funcionan bajo la siguiente tabla de verdad.

a	!a
F	V
V	F

Negación

a	b	a||b
F	F	F
F	V	V
V	F	V
V	V	V

Union

a	b	a&&b
F	F	F
F	V	F
V	F	F
V	V	V

Conjuncion

Al igual que sucede con los operadores aritméticos, los operadores lógicos y los relacionales tienen su precedencia. Ninguno tiene mayor precedencia que un operador matemático. Es esa la razón por la cual el último ejemplo no requiere de los paréntesis.

La siguiente es la tabla de precedencia para estos operadores:

!	mayor
> >= < <=	
== !=	
&&	
||	menor

Tabla de precedencia

Ejemplos:

-Se desea seleccionar aquellos números que se encuentren comprendidos entre 1 y 100. Las siguientes expresiones son equivalentes:

```
En_rango = numero > 0 && numero < 101
En_rango = numero >= 1 && numero <= 100
En_rango = (numero > 0) && (numero <= 100)
```

En este caso la variable `en_rango` contendrá un valor distinto de cero si la variable número contiene algún valor comprendido en el rango válido. De no ser así, entonces se le asignará un cero. Debe observarse que la expresión de relación contiene dos veces la variable número y no una como se expresaría en un lenguaje matemático puro. Esto es, un operador de relación o un operador lógico (a excepción de la negación) es un operador binario. Requiere de dos operandos, y no acepta un tercero. Por tanto la expresión siguiente es incorrecta:

```
En_rango = 1 <= número <= 100
```

- Se desea dar un incremento de salario a los empleados que sean del sexo femenino con más de 5 años de servicio y a los varones con más de 7 años de labor dentro de la empresa. También están incluídas en el aumento de salario aquellas personas que no hayan faltado a sus labores en el último año de servicio.

La expresión que definiría si un empleado cumple con los requisitos suponiedo las siguientes variables :

`annos_serv`	años de servicio en la empresa
`sexo`	sexo del trabajador
`faltas`	número de faltas en el año laborable

sería:

```
annos_serv > 5 && sexo == 'f' || annos_serv > 7 && sexo == 'm' || faltas == 0
```

Esto es lo mismo que tener tres expresiones separadas y unirlas despues a través del operador de OR.

```
(annos_serv>5  &&  sexo=='f') || (annos_serv>7  &&  sexo=='m') || (faltas == 0)
```

Los paréntesis no son necesarios dado que el compilador interpretará correctamente la precedencia de los operadores de relación. Sin embargo el uso de los paréntesis aumenta la claridad para el programador que realizará el mantenimiento posterior. Para el compilador no existirá diferencia alguna y ni siquiera correrá más lento de lo que lo haría si no se hubiesen usado los paréntesis en el programa fuente.

IMPLANTACION DE LA FUNCIÓN OR EXCLUSIVA

La función OR que se ha estudiado hasta el momento define que la expresión será verdadera si un miembro u otro o ambos son verdaderos. No siempre es deseable tener tal situación. En realidad a veces es necesario implantar funciones que cumplan con la siguiente tabla de verdad:

a	b	a⊕b
f	f	f
f	v	v
v	f	v
v	v	f

La función ha quedado definida como verdadera cuando uno solo de los miembros es verdadero y falso cuando ambos son falsos o ambos son verdaderos. Por tanto implantar esta o cualquier tabla de verdad solo requiere que se definan aquellos casos en los cuales la función se tomará verdadera. Esto se logra tomando la multiplicación de los términos que definan al producto como verdadero.

El primer término que produce un verdadero se encuentra en la condición en la cual a es falsa y b es verdadera. Esto nos define a la primera parte de la expresión como :

```
!a && b
```

El segundo término, en el cual se produce un verdadero, se obtiene de la misma manera. Este se volverá verdadero cuando a sea verdadero y b sea falso.:

```
a && !b
```

La expresión total quedaría así:

```
!a && b || a && !b
```

El procedimiento seguido en este ejemplo puede ser aplicado a cualquier tabla de verdad, pero debe tomarse en cuenta que no siempre se obtendrá una función con el menor número de productos. Muy

probablemente el lector ya habrá relacionado las *tablas de verdad* de este apartado con las *tablas de decisión* del capítulo uno. En realidad ambas cosas son lo mismo pero con un enfoque ligeramente distinto. La métodología que se expuso para obtener una función booleana, es perfectamente aplicable para obtener la función que regirá a un renglón de acciones de la tabla de decisión. El tema de la reducción de minitérminos (productos) queda fuera de los alcances de estos apuntes pero existe abundante información al respecto en los libros de *Diseño lógico* o de *Álgebra booleana.*

II.2.4 La estructura WHILE

La estructura while que ha continuación se estudiará es la misma que se ha visto en el capítulo uno de estos apuntes. En ella se evalúa primero una condición y si resulta verdadera entonces se ejecuta la o las instrucciones que se encuentren a continuación de la condición. De lo contrario se ejecutará la siguiente instrucción que no forme parte del bloque de instrucciones del while.

La forma general del while es:

while (condición)
sentencia;

Donde:
condición: es una expresión relacional o lógica.
sentencia : es cualquier sentencia válida en C.

Programa:
Hacer un programa que calcule los segundos transcurridos desde las cero horas hasta una hora definida por el usuario. Úsese un formato horario de 24 horas

Solución:

```
/* Ejemplo 19 */
/* 16/11/92 */
/* Cálculo de los segundos de un día */
/* Se supone un formato horario de 24 horas */

main()
{
      long int hora,minutos,segundos;


      printf("\n Cálculo de los segundos de un día");
      hora = 25;
      printf("\n Digite la hora y posteriormente los minutos [0-24 hrs]");
      while ( hora < 0 || hora > 24 )
            scanf("%i",&hora);
      printf("\n Hora:  %2i:",hora);
      minutos = 65;
      while ( minutos < 0 || minutos > 59 )
               scanf("%i",&minutos);
      segundos = 60* (minutos +hora*60);
      printf("Los segundos transcurridos son: %ld", segundos);
}
```

Resultados:

```
C:\>EJEM19

 Cálculo de los segundos de un dia
 Digite la hora y posteriormente los minutos [0-24 hrs]12

 Hora:  12:33
Los segundos transcurridos son: 45180
```

Observaciones:

Los ciclos while han sido utilizados para validar los datos de entrada. En primer lugar se coloca la variable de control del while en un estado tal que permita la entrada al ciclo. Si lo que se desea es que el ciclo while continue leyendo datos mientras estos sean incorrectos, entonces el dato antes de entrar al ciclo debe ser un dato incorrecto. Por ejemplo, si deseamos leer la hora en un formato de cero a veinticuatro horas, entonces para asegurarnos de que se entrará en el ciclo while colocamos la hora con el valor de 25 que es a todas luces inválido para el programa pero que satisface la condición de repetición de while :
hora < 0 || hora > 24. Tan pronto como el **scanf()** tome un dato correcto el ciclo terminará y continuará la ejecución del resto del programa.

Básicamente existen tres formas de controlar un ciclo. El primero es el que se ha visto en el ejemplo. A un while de este tipo se le conoce como un *ciclo controlado por condición.* Una segunda forma de controlar un ciclo while es *por contador* y la tercera es *por bandera.* Todas ellas serán vistas en los siguientes ejemplos.

En el ejemplo mostrado anteriormente, el ciclo while controla una sola sentencia. Cuando deseamos controlar un bloque completo de instrucciones entonces usamos las llaves para encerrar a todas las instrucciones que irán unidas conformando dicho bloque. Es importante recalcar que en la llave derecha que cierra al bloque no se coloca ningun punto punto decimal.

La forma general de un ciclo while para un bloque de instrucciones es la siguiente:

```
while (condicion)
{
        primera sentencia;
        segunda sentencia;
        .
        .
        .
        ultima sentencia;
}
```

Las llaves pueden ser colocadas con la sangría mostrada o siguiendo cualquier otro formato; por ejemplo con la llave izquierda inmediatamente después del paréntesis que cierra la condición. Es recomendable que en los programas se utilice un formato tal que permita ver a primera vista cual es el ámbito de control de una sentencia. En el formato sugerido es fácil observar donde inicia y donde finaliza el while.

Programa:
Hacer un programa que dibuje el siguiente triángulo:

```
1
12
123
1234
12345
```

hasta un número definido por el usuario y que puede llegar hasta 20.

Pseudocódigo:

```
DECLARA limite,renglon,columna,continua
ESCRIBE TITULAR(" Dibuja un triangulo")
HAZ MIENTRAS ( Se desee continuar )
      LEE (numero de renglones a dibujar)
      HAZ MIENTRAS (No se dibujen todos los renglones)
            EJECUTA UN salto de linea
            INICIALIZA numero EN LA PRIMERA COLUMNA
            HAZ MIENTRAS ( No se escriba todos los números del renglon)
                  INCREMENTA numero
                  ESCRIBE (numero)
            FIN DEL HAZ
      FIN DEL HAZ
FIN DEL HAZ
ESCRIBE("Desea dibujar otro triangulo?[S/N]")
LEE ( continuar )
```

Solución:

```
/* Ejemplo 20 */
/* 16/11/92 */
/* Programa que dibuja un triangulo */

main()
{
   int limite,renglon,columna;
   char continua;

   printf("\n Dibuja un triangulo\n\n");

   /* Ciclo de control del programa */
   continua = 's';
   while (continua != 'N' )
   {
      /* Lectura del numero de renglones a dibujar */
      printf("\n Digite el limite [0-20]");
      limite = -1;
      while ( limite < 0 || limite > 20 )
            scanf("%i",&limite);
      printf("\n Triangulo con %2i renglones",limite);
```

```
        /* Control de la impresion de cada renglon */
        renglon=0;
        while ( renglon < limite )
        {
              renglon++;
              printf("\n");
              columna=0;

              /* ESte ciclo imprime cada número del renglon */
              while (columna < renglon)
              {
                    columna++;
                    printf("%i",columna);
              }
        }


        printf ("\n\n Desea dibujar otro triangulo?[S/N]");
        continua = toupper(getche());
    }
}
```

Resultados:

```
C:\>EJEM20

 Dibuja un triangulo


 Digite el limite [0-20]5

 Triangulo con  5 renglones
1
12
123
1234
12345

 Desea dibujar otro triangulo?[S/N]N
```

Observaciones:

El programa utiliza los ciclos while de diversas formas. El primero que encontramos y que está controlado por la variable `continua`, nos permite controlar la ejecución total del programa. Este ciclo nos permite repetir la ejecución del programa tantas veces como el usuario desee repetirlo. Ya no es necesario volver a cargar desde el sistema operativo la línea de comando que activa el programa. Por otra parte esta ejecución no puede ser determinada de antemano (razón por la cual no se utiliza un ciclo for sino que depende de un evento aleatorio (la voluntad del usuario).

La variable `continua` es inicializada con el valor de 's' aunque en realidad hubiese bastado con cualquier caractér que fuera distinto de la letra N. La condición de salida del ciclo establece que mientras el caractér almacenado en la variable `continua` sea distinto de la letra N mayúscula el programa estará

ejecutándose. Si el usuario desea continuar sin tomarse la molestia de buscar la S podrá hacerlo pulsando cualquier caractér. Para salir del programa es necesaria la letra N aún cuando es intrascendente si el teclado está activo para las mayúsculas o las minúsculas. La función `toupper()` realiza la conversión del caractér a su equivalente en mayúsculas.

El siguiente while realiza la lectura de la variable `limite` tal y como fué expuesto en el ejemplo anterior con la finalidad de obtener un valor que servirá para determinar el número de renglones y columnas que se dibujarán. El número será positivo y menor a un máximo establecido por el programador.

El `printf()` que se encuentra después del ciclo while funciona como una verificación visual de lo que el usuario ha introducido en la máquina. De esta manera el usuario tiene una forma de verificar que se ha leído correctamente lo que ha sido tecleado.

El while controlado por la variable `renglon` es un ciclo controlado por un contador. Este ciclo se repetirá en tanto la variable no sea mayor al `limite`. Cada vez que se ejecute este ciclo el programa dará como resultado el despliegue de un renglón completo. Incrementando el número de renglones en cada ejecución.

El ciclo while más interno se efectuará en forma anidada. Cada vez que se llegue a esta sentencia se desplegarán los números que conforman cada renglón.

Estos dos últimos ciclos están siendo controlados por sus respectivos contadores. Y en el ciclo más interno el límite superior del contador está variando dinámicamente.

Según se expuso en un apartado anterior, este programa bien podía ser implantado por ciclos for, sin embargo es más natural escribirlo con la sentencia while que sugiere de una manera más clara que los ciclos se ejecutarán en tanto no se cumpla una condición. En este caso, la llegada a un valor por parte de un contador.

II.2.5 La estructura IF-ELSE

Una computadora no es tan importante por el hecho de que sea capaz de realizar millones de instrucciones por segundo. Ni por que tenga capacidad para almacenar millones, billones o millones de millones de caracteres en sus dispositivos de almacenamiento. Si bien son características que la convierten en un instrumento poderoso, la característica que diferencia a una computadora de cualquier otra máquina es la capacidad para tomar decisiones.

Las decisiones que una computadora es capaz de tomar vienen dadas por las expresiones de relación y las expresiones definidas en un punto anterior. Una decisión se realiza inquiriendo por el valor que una expresión posee. Si la expresión es evaluada con un valor de verdadero entonces se ejecuta una sentencia o un bloque de ellas. En caso de que la expresión resulte falsa pueden suceder tres casos:

1.- Se continua la ejecución del programa sin ejecutar ninguna de las instrucciones contenidas en el bloque verdadero del if
2.- Se ejecuta una sentencia correspondiente a un bloque falso de la decisión
3.- Se ejecuta un bloque de instrucciones.

En cualesquiera que sea el caso, tan pronto como se termina la ejecución de las instrucciones del if se continua ejecutando la siguiente instrucción colocada inmediatamente después del if.

La forma general de una sentencia if es:

```
if (condición)
        sentencia que se ejecuta si la condición es verdadera;
else
        sentencia que se ejecuta si la condición es falsa;
```

O bien en su forma de bloques:

```
if ( condición)
{
        primer sentencia del bloque verdadero;
        segunda sentencia;
        .
        .
        .
        ultima sentencia del bloque verdadero;
}
else
{
        primera sentencia del bloque falso;
        segunda sentencia;
        .
        .
        .
        ultima sentencia del bloque falso;
}
```

Como se mencionó anteriormente, el bloque falso (else) es opcional. No es necesario que una decisión conste de un bloque falso.

Programa:
En una nave espacial se dispone de un aparato para medir el diámetro de los planetas. De tal forma que es posible obtener el diámetro mayor y menor de un planeta ovoide. Se desea un programa que calcule el volúmen del mismo si la fórmula es:

$$v=4/3\ \pi\ a^2 b$$

Donde
a: diámetro mayor
b: diámetro menor

Solución:

```
/* Ejemplo 21 */
/* 17/11/92 */
/* Proyecto:  Volúmen de un planeta */

main()
{
   float diam_may,diam_men,temporal,volumen;
   const float pi=3.1415927;
   char continua;

   printf("\n Volúmen de un planeta\n\n");
```

```
   /* Ciclo de control del programa */
   continua = 's';
   while (continua != 'N' )
   {
      /* Lectura de los diametros de los planetas */

      diam_may = -1;
      while ( diam_may < 0 )
      {
            printf("\n Digite el primer diámetro [kms]: ");
            scanf("%f",&diam_may);
            if (diam_may <0)
                  printf("\n La cantidad debe ser positiva");
      }


      diam_men = -1;
      while ( diam_men < 0 )
      {
            printf("\n Digite el segundo diámetro [kms]: ");
            scanf("%f",&diam_men);
            if (diam_men <0)
                  printf("\n La cantidad debe ser positiva");
      }

      /* cálculo del volúmen */
      if (diam_may < diam_men)
      {
            temporal = diam_may;
            diam_may = diam_men;
            diam_men = temporal;
      }
      volúmen = (4.0/3.0)*pi*diam_may*diam_may*diam_men;

      /* Impresion de resultados */
      printf("\n\n\t\t    Tabla de resultados ");
      printf("\n  \t\t    ====================");
      printf("\n\n \t Las caracteristicas del planeta son: ");
      printf("\n\n\t Diámetro mayor      : %9.3f Kms.",diam_may);
      printf("\n  \t Diámetro menor      : %9.3f Kms.",diam_men);
      printf("\n  \t Volúmen del planeta : %9.3f Kms cúbicos", volumen);

      printf ("\n\n Desea otro cálculo?[S/N]");
      continua = toupper(getche());
   }
}
```

Resultados:

```
C:\>EJEM21
```

```
Volúmen de un planeta


Digite el primer diámetro [kms]: -12

La cantidad debe ser positiva
Digite el primer diámetro [kms]: 100

Digite el segundo diámetro [kms]: -45

La cantidad debe ser positiva
Digite el segundo diámetro [kms]: 110.01


                    Tabla de resultados
                    ===================

        Las características del planeta son:

        Diámetro mayor      :   110.010 Kms.
        Diámetro menor      :   100.000 Kms.
        Volúmen del planeta : 5069358.000 Kms cúbicos


Desea otro cálculo?[S/N]S

Digite el primer diámetro [kms]: 110.01

Digite el segundo diámetro [kms]: 100


                    Tabla de resultados
                    ===================

        Las características del planeta son:

        Diámetro mayor      :   110.010 Kms.
        Diámetro menor      :   100.000 Kms.
        Volúmen del planeta : 5069358.000 Kms cúbicos

Desea otro cálculo?[S/N]N
```

Observaciones:

En este ejemplo se está utilizando la variable pi como una constante. El valor de pi es de hecho una contante cuyo valor no debe cambiar a lo largo del programa, aunque dicho programa sea de 10000 líneas. Para asegurarnos de que *no se intentará alterar* dicho valor lo asignamos a una ***constante***. Por otra parte esto nos trae la facilidad de que si se desea aumentar la precisión del valor, el cambio solo deberá realizarse en una línea del programa y no en cada aparición de dicho valor.

La validación ahora es más completa ya que si el usuario comete un error durante la captura de la información el programa informará de inmediato cual es la condición que ha provocado el rechazo del dato y repite explícitamente la petición.

El programa no obliga a que el usuario sepa cuál es el diámetro mayor y cuál es el menor, no exige que se introduzcan en un orden específico. En forma autónoma el programa realizará una validación interna sobre el dato mayor y el menor, de tal manera que tan pronto como salga del if los datos estarán listos para ser procesados por la fórmula que calcula el volúmen.

En el ejemplo mostrado se han utilizado decisiones sin bloque falso. Cuando la condición evaluada es falsa todo el bloque verdadero es ignorado y se continúa la ejecución del programa.

Programa:
Un trabajador recibe su sueldo normal por las primeras 30 horas y se le paga 1 1/2 veces su sueldo normal por cada hora después de las primeras 30. Escriba un programa que calcule el pago

Pseudocódigo:

```
Cálculo de nómina

DECLARACION de los parámetros de sistema
      ESCRIBE(" Cálculo de la Nómina");
HAZ MIENTRAS (se desee ejecutar )
      LEE(nombre_del_empleado)
      LEE(horas_trabajadas)
      SI (no hay horas_extras )
            CALCULA sueldo
      SINO
            CALCULA horas_extras
            CALCULA compensacion
            CALCULA sueldo
      FIN DE SI
            SI (no hay horas_extras)
            ESCRIBE(Sueldo)
      SINO
            ESCRIBE(Sueldo,Compensancion)
      FIN DEL SI
      ESCRIBE(" Desea otro cálculo?[S/N]")
      LEE (respuesta)
FIN DEL HAZ
```

Solución:

```
/* Ejemplo 22 */
/* 16/11/92 */
/* Cálculo de nómina */

main()
{
   /* Bloque de declaraciones  */
   float sueldo, horas, h_extras,compensacion;
   char continua;
   char nombre[80];
```

```
/*  Parámetros de sistema     */
const float max_horas  = 50;
const float sal_normal = 11000;
const float fac_extras = 1.5;
const int   lim_extras = 30;


printf("\n\t\t Cálculo de la Nómina");
printf("\n\t\t ====================\n\n");

/* Ciclo de control del programa */
continua = 's';
while (continua != 'N' )
{
   /* Lectura del nombre del empleado */
   printf("\n Nombre del empleado: ");
   nombre[0] = '\0';
   while ( nombre[0] == '\0' )
   {
         /* Esta funcion no es estándar. Tiene por finalidad
            limpiar de caracteres basura a la memoria del entrada */
         flushall();

         gets(nombre);
         if (nombre[0] == '\0')
               printf("\n Digite el nombre del empleado\n");
   }


   /* Lectura del número de horas trabajadas */
   horas = 0.0;
   while (horas <= 0 || horas > max_horas)
   {
         printf("\n Digite el número de horas trabajadas: ");
         scanf("%f",&horas);
         if (horas <= 0  )
               printf("\n El número de horas debe ser positivo ");
         else
               if (horas > max_horas)
                     printf ("\n El maximo número de horas es %5.1f"
                     ,max_horas);
   }

   /* Cálculo del salario */
   sueldo = sal_normal;
   h_extras = 0;
   if (horas <= lim_extras )
   {
         sueldo = sal_normal*horas;
   }
   else
```

```
        {
                h_extras = horas - lim_extras;
                compensacion= h_extras*sal_normal*fac_extras;
                sueldo = sal_normal*lim_extras;
                sueldo = sueldo + compensacion;

        }

        /* Despliege de resultados   */
        printf (" \n\n Salario correspondiente a %s : ",nombre);
        if (horas <= lim_extras)
        {
                printf(" \n\n\t Sueldo: %9.2f",sueldo);
        }
        else
        {
                printf(" \n\n\t Sueldo Normal: %9.2f",sal_normal*lim_extras);
                printf("   \n\t Horas extras : %9.2f",h_extras);
                printf("   \n\t Compensancion: %9.2f",compensacion);
                printf("   \n\t                ========");
                printf("   \n\t Total a pagar: %9.2f",sueldo);
        }

        printf ("\n\n Desea otro cálculo?[S/N]");
        continua = toupper(getche());
    }
}
```

Resultados:

```
                Cálculo de la Nómina
                ====================


Nombre del empleado: Ernesto Peñaloza Romero

Digite el número de horas trabajadas: 0

El número de horas debe ser positivo
Digite el número de horas trabajadas: 51

El maximo número de horas es  50.0
Digite el número de horas trabajadas: 1


Salario correspondiente a Ernesto Peñaloza Romero :

        Sueldo:  11000.00

Desea otro cálculo?[S/N]=
```

```
Nombre del empleado:Lucero Hogaza Leon

Digite el número de horas trabajadas: 40


Salario correspondiente a Lucero Hogaza Leon :

        Sueldo Normal: 330000.00
        Horas extras :     10.00
        Compensancion: 165000.00
                       ==========
        Total a pagar: 495000.00

Desea otro cálculo?[S/N]n
```

Observaciones:

El programa que se ha escrito tiene algunas conductas mucho más interesantes. El programa se encuentra tomando algunas decisiones que modifican la apariencia del reporte de salida.

Si el trabajador percibe solo su salario normal este es indicado a través de una sola línea en el reporte de salida. Si por el contrario, el trabajador percibirá una compensación por horas extras, el programa desplegará un reporte desglosado de las percepciones del trabajador.

Debe hacerse especial mención de la función `flushall()` que se encuentra antes de la lectura del nombre del empleado. Dicha función no es estándar y deberá buscarse su eqiuvalente en la implantación que usted utilice. Dentro del entorno de C todas las entradas son enviadas a una memoria intermedia (buffer) que retiene los caracteres leídos hasta que estos son requeridos por alguna función de entrada. La función `gets()` lee la cadena hasta encontrar un salto de línea (\n ó <ENTER>). La cadena es almacenada en el arreglo especificado en la función y el salto de línea es convertido en un nulo (\0). Pero el salto de línea no es eliminado de la memoria intermedia. La función **flushall()** elimina este o cualquier otro caractér antes de realizar la lectura con la función **gets()**. De no existir esta función el salto de línea retenido en la primera lectura será asignado como un nulo en la segunda lectura del `gets()` y el mensaje de error del if de validación será desplegado.

Programa:

Escribir un programa que obtenga todos los números primos desde el número 3 hasta un número definido por el usuario.

Solución:

Recordando que un número primo es aquel que solamente es divisible entre sí mismo y entre el número uno, la solución al problema se obtiene dividiendo al **numero** que no interesa entre todas las cantidades existentes entre 1 y el **numero**. Cuando se encuentra que el residuo de una de esas divisiones es cero y el divisor no es el uno o el **numero**, entonces es posible afirmar que el **numero** no es primo.

```
/* Ejemplo 23 */
/* 16/11/92 */
/* Determina si un número es primo */
```

```
main()
{
   int numero, divisor, primo;
   char continua;

   printf("\n Determina números primos\n\n");

   /* Ciclo de control del programa */
   continua = 's';
   while (continua != 'N' )
   {
      /* Lectura del número limite */
      printf("\n Digite el número : ");
      numero = -1;
      while ( numero < 0 )
            scanf("%i",&numero);

      /* bandera de control para el ciclo while */
      primo = 1;
      divisor=numero-1;

      while ( primo && (divisor > 1) )
      {
            if (numero % divisor )
            {
                  divisor--;
            }
            else
            {
                  primo=0;
            }
      }
      if (primo)
            printf("\n %i es un número primo",numero);
      else
            printf("\n %i no es un número primo",numero);


      printf ("\n\n Desea otro cálculo?[S/N]");
      continua = toupper(getche());
   }
}
```

Resultados:

```
C:\COMP\TC2>ejem23

 Determina números primos


 Digite el número : 1
```

```
1 es un número primo

Desea otro cálculo?[S/N]s
Digite el número : 1000

1000 no es un número primo

Desea otro cálculo?[S/N]s
Digite el número : 9

9 no es un número primo

Desea otro cálculo?[S/N]n
```

Observaciones:

El programa está basado en la definición de un número primo. Para ello se utiliza el ciclo while que efectúa una a una las divisiones del número entre todos los enteros comprendidos entre él y el uno. El ciclo puede detenerse de dos formas. Una es cuando se encuentra que el divisor ha alcanzado el valor de uno ya que no ha sido posible encontrar una división con residuo igual a cero. Este caso hará que la variable **primo** se mantenga encendida. El número, por definición, ha sido encontrado primo. La otra forma en la cual el ciclo se detiene es precisamente cuando se encuentra una división con residuo igual a cero. Se ha encontrado que el número no es primo, se apaga la variable y en la siguiente pregunta de condición que efectúa el while el ciclo concluye sin terminar de ejecutar todo el resto de las divisiones.

Este es un ejemplo de ciclo controlado por bandera. El ciclo concluye cuando una bandera indica que la tarea ha sido concluida y que se puede dar por satisfecho el proceso de iteración. El programa bien podría ser escrito sin la necesidad de utilizar el if anidado en el ciclo while. Dicho programa debería ser escrito de la siguiente manera

Solución alternativa:

```
/* Ejemplo 24 */
/* 17/11/92 */
/* Determina si un número es primo
   Solución alternativa              */

main()
{
   int numero, divisor, primo;
   char continua;

   printf("\n Determina números primos\n\n");

   /* Ciclo de control del programa */
   continua = 's';
   while (continua != 'N' )
   {
      /* Lectura del número limite */
      printf("\n Digite el número : ");
      numero = -1;
      while ( numero < 0 )
            scanf("%i",&numero);
```

```
        /* bandera de control para el ciclo while */
        primo = 1;
        divisor=numero-1;

        while ( primo && (divisor > 1) )
        {
              /* aqui esta la diferencia entre ambos programas */
              primo =numero % divisor;
              divisor--;
        }
        if (primo)
              printf("\n %i es un número primo",numero);
        else
              printf("\n %i no es un número primo",numero);


        printf ("\n\n Desea otro cálculo?[S/N]");
        continua = toupper(getche());
    }
}
```

Un bloque correspondiente a cualquier rama de un if puede contener cualquier sentencia, y esto incluye a otra sentencia if. Al igual que en los ciclos for y los ciclos while, el anidamiento de sentencias es permitido siempre y cuando el ámbito de control de las sentencias anidadas no se translapen.

Programa:
Hacer un programa que lea los tres coeficientes de una ecuación cuadrática y determine sus raíces. El programa deberá informar si las raíces son iguales o diferentes así como si son de solución real o imaginaria.

Solución:
Las raíces de una ecuación cuadrática se obtienen resolviendo la fórmula general

$$x_1, x_2 = \frac{-b \pm (b^2 - 4ac)^{1/2}}{2a}$$

El radical $b^2 - 4ac$ puede tener tres posibles valores los cuales indican el tipo de raíz que posee la curva en estudio. Estos tres valores son listados en la siguiente tabla:

$b^2 - 4ac < 0$ Las raíces son imaginarias y distintas
$b^2 - 4ac = 0$ Las raíces son reales e iguales
$b^2 - 4ac > 0$ Las raíces son reales y distintas

Solución:

```
/* Ejemplo 25 */
/* 19/Nov/92 */
/* Raices de una ecuación cuadrática a través
   de la fórmula general                    */
#include <math.h>
```

```
main()
{
   float coef_a, coef_b, coef_c,  raiz_1, raiz_2;
   float imaginaria;
   double radical;
   char continua;
   int correcto;

   printf("\n Solución de ecuaciones cuadráticas\n\n");

   /* Ciclo de control del programa */
   continua = 's';
   while (continua != 'N' )
   {
      printf(" La forma general de una ecuación cuadrática es:");
      printf(" ax^2 + bx + c = 0 \n");

      /* Lectura de primer coeficiente */
      coef_a = 0.0;
      printf("\n Digite el coeficiente 'a': ");
      scanf("%f",&coef_a);
      correcto = 1;
      if (coef_a == 0.0)
      {
            printf("\nEste coeficiente debe ser distinto a cero");
            correcto = 0;
      }

      if (correcto)
      {
            /*   lectura del segundo coeficiente */
            coef_b = 0.0;
            printf("\n Digite el coeficiente 'b': ");
            scanf("%f",&coef_b);

            /*   lectura del término independiente */
            coef_c = 0.0;
            printf("\n Digite el término independiente 'c': ");
            scanf("%f",&coef_c);

            /* Determinación del tipo de raiz  */
            radical = (coef_b*coef_b - 4.0*coef_a*coef_c);
            if ( radical < 0.0 )
            {
                  printf("\n Las raices son imaginarias");
                  radical = sqrt( abs(radical));
                  imaginaria =  radical / (2*coef_a);
                  raiz_1 = -coef_b / (2*coef_a);

                  printf("\n X(1) = %8.3f + %8.3f j",raiz_1, imaginaria);
                  printf("\n X(2) = %8.3f - %8.3f j",raiz_1, imaginaria);
            }
            else
```

```
                if ( radical == 0.0 )
                {
                     printf("\n Las raices son reales e iguales");
                     raiz_1 =  -coef_b / (2*coef_a);
                     printf ("\n X(1) = X(2) = %8.3f",raiz_1);
                }
                else
                {
                     printf("\n Las raices son reales y distintas");
                     raiz_1 = (-coef_b + sqrt(radical)) / (2*coef_a);
                     raiz_2 = (-coef_b - sqrt(radical)) / (2*coef_a);
                     printf("\n X(1) = %8.3f",raiz_1);
                     printf("\n X(2) = %8.3f",raiz_2);
                }
        }


        printf ("\n\n Desea otro cálculo?[S/N]");
        continua = toupper(getche());
    }
}
```

Resultados:

```
C:\COMP\TC2>ejem25

 Solución de ecuaciones cuadráticas


 La forma general de una ecuación cuadrática es: ax^2 + bx + c = 0

 Digite el coeficiente 'a': 1

 Digite el coeficiente 'b': 4

 Digite el término independiente 'c': 4

 Las raices son reales e iguales
 X(1) = X(2) =   -2.000

 Desea otro cálculo?[S/N]s

 La forma general de una ecuación cuadrática es: ax^2 + bx + c = 0

 Digite el coeficiente 'a': 5

 Digite el coeficiente 'b': 1

 Digite el término independiente 'c': 2
```

```
Las raíces son imaginarias
X(1) =   -0.100 +     0.624 j
X(2) =   -0.100 -     0.624 j

Desea otro cálculo?[S/N]s
La forma general de una ecuación cuadrática es: ax^2 + bx + c = 0

Digite el coeficiente 'a': 1

Digite el coeficiente 'b': 4

Digite el término independiente 'c': 2

Las raíces son reales y distintas
X(1) =   -0.586
X(2) =   -3.414

Desea otro cálculo?[S/N]n
```

Observaciones:

En las primeras líneas se ha hecho referencia a una librería estandar de C. La referencia se ha realizado con el `#include <math.h>` que posee los prototipos de las funciones matemáticas que C puede manejar. En el ejemplo que hemos realizado se utiliza la función `sqrt()` la cual obtiene la raíz cuadrada del valor enviado como argumento.

Una de las diferencias más notables de este programa en cuanto a la forma en la cual se realiza una validación es que, en los otros programas, el error era atrapado y el usuario no podía salir del segmento de captura a menos que proporcionara un argumento válido.

En el ejemplo recién expuesto no se atrapa al error de esta manera. Si se detecta que el coeficiente a es cero, es decir, si no se trata de una ecuación de segundo orden, es desplegado un error y nada que exista con posterioridad es ejecutado. La variable `correcto` funge como un indicador del estado que guarda el programa; permanecerá encedida (valor de 1) cuando el coeficiente sea válido. La variable encendida permitirá la ejecución del resto del programa dado que todo es correcto. Si el coeficiente a es detectado como no válido, la variable `correcto` es apagada y el segmento de código que calcula las raíces no es ejecutado.

Esta técnica puede ser aplicada en cualquier programa en donde se deba validar un punto crucial que pueda hacer abortar el programa o entregar resultados erróneos. Por ejemplo, antes de emitir un reporte es necesario verificar si la impresora está en línea, de no ser así se envía un mensaje de error y se omite la ejecución del programa generador del reporte. O bien cuando se tengan que validar una serie de eventos independientes, como la apertura de varios archivos antes de proceder a ejecutar un proceso. Únicamente una variable será necesaria para indicar que el programa se está ejecutando correctamente, de no ser así el if se encargará de aislar el código que no pueda ser ejecutado debido al error.

Existen muchas alternativas al modelo planteado. El que se ha presentado representa una primera aproximación a la forma como se pueden manipular los errores.

Dentro de la rama verdadera del if(correcto) se encuentra anidado una segunda sentencia if. A su vez dentro de la rama falsa de este segundo if se encuentra un tercer if anidado. Es importante que se observe la manera como se delimita el alcance de cada una de las sentencias if. El else siempre hace referencia al if más cercano que haya cerrado inmediatamente antes su rama verdadera.

El estándar ANSI especifica que se deben permitir anidamientos de por los menos 15 niveles.

II.2.5 La estructura SWITCH

En muchos programas se desea hacer comparaciones relativas a una sola variable que puede almacenar un número finito de casos conocidos. En base al valor de dicha variable se ejecutarán ciertos procedimientos o no.

Una serie de if anidados bien podrían ayudar en este caso, sin embargo C proporciona la estructura de decisión múltiple a través de la sentencia switch cuya forma general es :

```
switch(expresión)
{
        case constate1:
                bloque 1 de sentencias;
                break;
        case constate2:
                bloque 2 de sentencias;
                break;
        case constate3:
                bloque 3 de sentencias;
                break;
        .
        .
        .
        case constaten:
                bloque n de sentencias;
                break;
        default:
                bloque n+1 de sentencias;
}
```

La sentencia switch comprobará la igualdad de la expresión con cada una de las constantes enunciadas en los case. El orden de evaluación será estrictamente secuencial y tan pronto como se encuentre una igualdad entonces se procederá a ejecutar las sentencias asociadas con el case en cuestión. La ejecución de las sentencias continuará en forma secuencial hasta encontrar la sentencia break o hasta alcanzar el final del ámbito de la sentencia switch.

En el caso de que al evaluar no se encuentre correspondencia de igualdad con el case en turno, todas las sentencias que se encuentren relacionadas con él no serán ejecutadas y se saltarán hasta alcanzar el siguiente case. Un case no posee significado por sí solo. Es decir, un case no puede existir fuera del ámbito de control de una sentencia switch.

Cuando se evalúen todos los case y no se encuentre ninguna correspondencia de igualdad, se ejecutarán entonces las sentencias asociadas al default.

La sentencia break, (break: romper. Alusión al rompimiento de la secuencia para saltar a otro punto del programa) es necesaria para que cada case ejecute solo las sentencias ubicadas dentro de su ámbito. Cuando dicha sentencia es encontrada el control pasa a la siguiente instrucción fuera del alcance del switch. Si no se incluye dicha sentencia *no se romperá el control* y se ejecutará también el siguiente bloque de instrucciones.

Programa:
Realizar un programa que simule una calculadora sobre las cuatro operaciones básicas. Primero se leerá el operando y posteriormente los dos operadores.

Solución:
Se está pidiendo una calculadora con notación prefija. Para realizar dicho programa se piden los datos que fungirán como operandos y posteriormente se realizará la selección del caractér que representa la operación.

```
/* Ejemplo 26 */
/* 20/Nov/92 */
/* Calculadora bajo notación polaca prefija      */


main()
{
   float oper_1, oper_2 ;
   float resultado;
   char operador;
   int correcto;
   /* La siguiente linea indica que la tecla ESCAPE tiene un valor de 27
      lo cual es verdadero para teclados ASCII */
   const int ESC=27;

   printf("\n\tCalculadora prefija");
   printf("\n\t==================\n\n");

   /* Ciclo de control del programa */
   operador = '';
   while (operador != ESC )
   {
      printf("\n\n Uso : <operador> <operando 1> <operando 2>");


      /* Lectura del operando        */
      printf("\n\n Operadores: <+> <-> </> <*> <ESC>: salir");
      printf("\n Digite operando: " );
      operador = getche();
      if (operador != ESC )
      {
         /* Se define si el operador fue correcto */
         correcto= operador == '+' || operador == '-';
         correcto= operador == '*' || operador == '/' || correcto;

         if (correcto)
         {

            /* Lectura de primer operando */
            oper_1 = 0.0;
            printf("\n Primer operando  : ");
            scanf("%g",&oper_1);
```

```
            /* Lectura del segundo operando*/
            oper_2 = 0.0;
            printf("\n Segundo operando : ");
            scanf("%g",&oper_2);

            /* Seleccion de la operacion a realizar */

            switch (operador)
            {
                case '+':
                  printf("\n Operación: SUMA");
                  resultado = oper_1 + oper_2;
                  printf("\n %g + %g = %g",oper_1,oper_2,resultado);
                  break;

                case '-':
                  printf("\n Operación: RESTA");
                  resultado = oper_1 - oper_2;
                  printf("\n %g - %g = %g",oper_1,oper_2,resultado);
                  break;

                case '*':
                  printf("\n Operación: MULTIPLICACION");
                  resultado = oper_1 * oper_2;
                  printf("\n %g * %g = %g",oper_1,oper_2,resultado);
                  break;

                case '/':
                  printf("\n Operación: DIVISION");
                  if (oper_2 !=0)
                  {
                      resultado = oper_1 / oper_2;
                      printf("\n %g / %g = %g",oper_1,oper_2,resultado);
                  }
                  else
                      printf ("\n La division entre cero no es válida");
            }
          }
          else
            printf("\n\n El operador no es válido ");
      }

   }
}
```

Resultados:

```
C:\COMP\TC2>ejem26
```

```
        Calculadora prefija
        ===================


Uso : <operador> <operando 1> <operando 2>

Operadores: <+> <-> </> <*> <ESC>: salir
Digite operando: +
Primer operando  : 12

Segundo operando : 10

Operación: SUMA
12 + 10 = 22

Uso : <operador> <operando 1> <operando 2>

Operadores: <+> <-> </> <*> <ESC>: salir
Digite operando:<ESC>
```

Obsevaciones:

La primera característica nueva que tiene el programa es la definición a través de una constante del valor de la tecla ESCAPE. En la mayoría de las implantaciones que usan el código ASCII para la representación de la información, el valor asociado a la tecla ESCAPE es el 27. Si su máquina no usa dicho código, basta con averiguar cual es el valor de representación para la tecla y sustituirlo en dicha línea.

Ahora el programa no preguntará de manera explícita si se desea dar por finalizado el proceso. En su lugar, el usuario tiene la libertad de elegir entre un operando o la orden de salida. La orden de salida se almacena en la misma localidad de salida en donde se almacenan los operadores, es decir , en la variable **operador**.

Por definición del problema, lo primero que debe leer el programa es el operador con el cual se va a trabajar. En este programa el operador es sometido a una extenso trabajo de validación. Tan pronto como es leído es necesario saber si se ordenó salir del programa, con el fin de terminar lo antes posible y no ocupar más al procesador. Si el operador no posee el valor asociado con la tecla ESCAPE entonces puede contener un operador válido. Así que se realiza la validación de dicho operador.

La validación del operador no se efectúa en una sola línea ya que resultaría en una expresión muy larga. En su lugar, dicha expresión se divide en dos partes y se asigna a la variable lógica **correcto**. En general las expresiones largas o complejas no son deseables en un programa ya que atentan contra la claridad del mismo.

Si el operador es válido entonces se realiza la selección de la operación solicitada por el usuario a través de la sentencia switch. En ella se pregunta por la variable operador para ver si cae en alguno de los casos listados dentro de las cláusulas case. Dado que **operador** es una variable caractér las constantes contra las cuales se compara la igualdad van encerradas entre apóstrofos.

Al final de cada bloque de sentencias se utiliza la sentencia break para alcanzar el fin de la sentencia switch. Esto es cierto con la excepción del último bloque de sentencias. Allí la sentencia no tiene utilidad ya que de cualquier forma ya se ha alcanzado el fin de la extensión del control de la sentencia switch.

La siguiente es una variación del mismo programa utilizando a la sentencia switch como validador.

```
/* Ejemplo 27 */
/* 20/Nov/92 */
/* Calculadora bajo notación polaca prefija
   Version 2: Validación con la sentencia switch */


main()
{
   float oper_1, oper_2 ;
   float resultado;
   char operador;
   /* La siguiente línea indica que la tecla ESCAPE tiene un valor de 27
      lo cual es verdadero para teclados ASCII */
   const int ESC=27;

   printf("\n\tCalculadora prefija");
   printf("\n\t=================\n\n");

   /* Ciclo de control del programa */
   operador = '';
   while (operador != ESC )
   {
      printf("\n\n Uso : <operador> <operando 1> <operando 2>");


      /* Lectura del operando       */
      printf("\n\n Operadores: <+> <-> </> <*> <ESC>: salir");
      printf("\n Digite operando: " );
      operador = getche();
      switch (operador)
      {

         /* Se define si el operador fue correcto */
         case '+':
         case '-':
         case '/':
         case '*':
            /* Lectura del primer operando */
            oper_1 = 0.0;
            printf("\n Primer operando  : ");
            scanf("%g",&oper_1);

            /* Lectura del segundo operando*/
            oper_2 = 0.0;
            printf("\n Segundo operando : ");
            scanf("%g",&oper_2);
```

```
        /* Seleccion de la operacion a realizar */

        switch (operador)
        {
            case '+':
              printf("\n Operación: SUMA");
              resultado = oper_1 + oper_2;
              printf("\n %g + %g = %g",oper_1,oper_2,resultado);
              break;

            case '-':
              printf("\n Operación: RESTA");
              resultado = oper_1 - oper_2;
              printf("\n %g - %g = %g",oper_1,oper_2,resultado);
              break;

            case '*':
              printf("\n Operación: MULTIPLICACION");
              resultado = oper_1 * oper_2;
              printf("\n %g * %g = %g",oper_1,oper_2,resultado);
              break;

            case '/':
              printf("\n Operación: DIVISION");
              if (oper_2 !=0)
              {
                  resultado = oper_1 / oper_2;
                  printf("\n %g / %g = %g",oper_1,oper_2,resultado);
              }
              else
                  printf ("\n La division entre cero no es valida");

        }
        break;
        /* si es la tecla escape */
      case 27:
        break;
      default:

        printf("\n Operador no válido");

      }

  }
}
```

Obsevaciones:

La variable `correcto` ha desaparecido así como el if que la utilizaba. En su lugar se utiliza una sentencia switch que se abre inmediatamente después de la lectura del operador.

Las primeras cláusulas case de esta sentencia están vacías. Si el operador digitado fuera por ejemplo el signo <->, la sentencia preguntaría por la igualdad del operador con '+', al no existir correspondencia saltaría al case con '-' . Allí si existe correspondencia de igualdad por lo que intenta ejecutar la sentencia asociada con dicha cláusula, Al no encontrar ninguna, ni una sentencia break, intenta ejecutar las sentencias asociadas al case '/' y posteriormente las asociadas al case '*'. En dicho case si encuentra sentencias y procede a ejecutarlas.

El bloque de sentencias tiene su punto de salida en el break anterior a la cláusula case 27. Por lo que al alcanzar la sentencia break, se rompe la ejecución y se da por finalizada la tarea del switch más externo continuando con la ejecución del programa.

Si la tecla que el usuario pulsó fue la tecla ESCAPE entonces el valor numérico que `operador` tiene almacenado es el 27, por lo tanto en el case se pregunta por el número 27 y no por el conjunto de caracteres '27' como podría pensarse en un momento dado.

Si `operador` no contiene ninguno de los valores definidos en las cláusulas case anteriores entonces se procede a ejecutar las sentencias definidas en la cláusula default, que en este caso es un aviso al usuario de que el operador solicitado no es válido.

La aplicación más común para una sentencia switch se encuentra en la realización de un menú de opciones de proceso.

Ejemplo:
Realizar el menú para un programa de altas, bajas, cambios, despliegues y extras de un programa de captura de nombre de empleados.

```
/* Ejemplo 28 */
/* 20/Nov/92 */
/* Ejemplo de la realización de un menú */

main()
{
   int opcion;
   const int ESC=27;



   /* Ciclo de control del programa */
   opcion = '';
   while (opcion != ESC )
   {
      clrscr();
      printf("\n\tCaptura de empleados");
      printf("\n\t=====================\n\n");
      printf("\n\t\t Altas");
      printf("\n\t\t Bajas");
      printf("\n\t\t Cambios");
      printf("\n\t\t Despliegues");
      printf("\n\t\t Extras");
      printf("\n\t\t <ESC> Salida");
      printf("\n\n\t Digite su opcion [A/B/C/D/E]");
      opcion=toupper(getch());
```

```
        switch (opcion)
        {
              case 'A':
                    /* Módulo de altas */
                    printf("\n Opcion: Altas");
                    /*
                    altas(); */
                    break;
              case 'B':
                    /* Módulo de Bajas  */
                    printf("\n Opcion: Bajas");
                    /*
                    bajas();  */
                    break;
              case 'C':
                    /* Módulo de Cambios */
                    printf("\n Opcion: Cambios");
                    /*
                    cambios()  */
                    break;
              case 'D':
                    /* Módulo de Despliegues */
                    printf("\n Opcion: Despliegues");
                    /*
                    despliegues();  */
                    break;
              case 'E':
                    /* Módulo de  Extras */
                    printf("\n Opcion: Extras");
                    /*
                    extras();  */
              default:
                    printf("\n\t Digite la letra inicial de su opcion");

        }
        if (opcion !=27)
        {
              printf("\n\n Pulse cualquier tecla para continuar");
              getch();
        }
    }
}
```

Resultados:

```
Captura de empleados
====================


        Altas
        Bajas
        Cambios
        Despliegues
        Extras
        <ESC> Salida

 Digite su opcion [A/B/C/D/E]
```

```
Despues de pulsar la letra <a> se visualizará:


        Captura de empleados
        ====================


                Altas
                Bajas
                Cambios
                Despliegues
                Extras
                <ESC> Salida

         Digite su opcion [A/B/C/D/E]
Opcion: Altas

Pulse cualquier tecla para continuar <ESC>
```

Observaciones:

Dado que es sólo un ejemplo no se incluyen los módulos que procesarían la información, pero si se muestra la línea que realizaría dicha llamada al módulo. En el ejemplo, el usuario digita la letra inicial de la opción que elija y terminará el programa pulsando la tecla <ESC>. Ya no aparece la pregunta sobre la continuación de proceso.

En este ejemplo no se ha desarrollado cada uno de los módulos que conformarán al sistema. Sólo se trata del módulo que realiza las llamadas a cada uno de los submódulos. En este programa se está realizando programación TOP-DOWN ya que el módulo principal ya está funcionando aunque los submódulos ni siquiera existan.

En general se puede decir que en una sentencia switch no pueden existir dos cláusulas case con la misma constante. Por otra parte solo se realiza una comprobación de igualdad, no es posible evaluar expresiones lógicas o relacionales dentro de una sentencia switch.

II.2.7 La estructura DO-WHILE

La última estructura de control que se estudiará es el do-while. Esta sentencia ejecuta primero la o las sentencias asociadas a su cuerpo de ejecución y hasta que ha finalizado con ellas realiza una pregunta para decidir si debe o no continuar con el ciclo.

La forma general de un do - while es

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

o su forma en bloque:

```
do
{
        primera sentencia;
        segunda sentencia;
        .
        .
        .
        ultima sentencia;
}       while (condicion);
```

Ejemplo:
Recodificar el ejemplo del menú con la sentencia do-while

```
/* Ejemplo 29 */
/* 20/Nov/92 */
/* Ejemplo de la realización de un menú
   Version con la estructura do-while */

main()
{
   int opcion;
   const int ESC=27;



   /* Ciclo de control del programa */

   do
   {
      clrscr();
      printf("\n\tCaptura de empleados");
      printf("\n\t==================\n\n");
      printf("\n\t\t Altas");
      printf("\n\t\t Bajas");
      printf("\n\t\t Cambios");
      printf("\n\t\t Despliegues");
      printf("\n\t\t Extras");
      printf("\n\t\t <ESC> Salida");
      printf("\n\n\t Digite su opcion [A/B/C/D/E]");
```

```
        opcion=toupper(getch());
        switch (opcion)
        {
              case 'A':
                    /* Módulo de altas */
                    printf("\n Opcion: Altas");
                    /*
                    altas(); */
                    break;
              case 'B':
                    /* Módulo de Bajas  */
                    printf("\n Opcion: Bajas");
                    /*
                    bajas();  */
                    break;
              case 'C':
                    /* Módulo de Cambios */
                    printf("\n Opcion: Cambios");
                    /*
                    cambios()  */
                    break;
              case 'D':
                    /* Módulo de Despliegues */
                    printf("\n Opcion: Despliegues");
                    /*
                    despliegues();  */
                    break;
              case 'E':
                    /* Módulo de  Extras */
                    printf("\n Opcion: Extras");
                    /*
                    extras();  */
              default:
                    printf("\n\t Digite la letra inicial de su opcion");

        }
        if (opcion !=27)
        {
              printf("\n\n Pulse cualquier tecla para continuar");
              getch();
        }


    } while (opcion != ESC );
}
```

Observación:

Tal y como se puede observar, el único cambio significativo se encuentra en la localización de la pregunta. Dado que ahora se encuentra al final de la estructura ya no ha sido necesario inicializar la variable **opcion**. Ademas ahora se asegura que el menú se ejecutará por lo menos una vez.

Preguntas:

1.- ¿Porqué un ciclo for no debe escribirse con alguna de sus tres condiciones vacias?
2.- ¿En que ocasiones debe utilizar un ciclo for?
3.- ¿Porqué no son recomendables los arreglos de mas de dos dimensiones?
4.- Explique claramente la diferencia entre una expresión de relación y una expresión lógica
5.- ¿Cómo se implanta una tabla de decisión?
6.- ¿Qué relación existe entre una expresión lógica y los árboles de decisión?
7.- ¿Cuales son las tres formas de implantar un ciclo while?
8.- ¿Qué sucede si dentro del cuerpo de un ciclo while, no se modifica la variable de control?
9.- Muchos programadores escriben alguna variante del siguiente código:

```
verdadero = 1;
while (verdadero)
{
      bloque de proceso;
      if  (condición_salida)
      {
            /* Salir del programa */
            exit( );
      }
      else
      {
            /* bloque de proceso */
      }
      bloque de proceso;
}
```

Según se expuso en el capítulo uno.
a) ¿Es esto correcto?
b) ¿Cuántos y cuales son los errores en el estilo de programación?
c) ¿Cómo pueden corregirse?
d) ¿ Cuales son los peligros potenciales de un código escrito de esta forma?

9.- Explique en base a los conceptos del capitulo uno, porque los ámbitos de control de un if no pueden ser transpasados impunemente. Es decir, porque no se puede realizar un salto arbitrario al interior de un if.
10.- ¿Porqué es valido usar una sentencia break dentro de una estructura switch?

II.3 Funciones

En el capítulo uno de estos apuntes se hizo incapié en la necesidad que existe de crear módulos que sean independientes entre sí. En la práctica, hay aún más razones de peso para modularizar las funciones que componen un programa.

Una función, en el sentido académico de la palabra, es un proceso con entradas y salidas bien definidas. Su implantación práctica también debe ir encaminada a la realización de bloques bien definidos en cuanto al proceso que realicen y los insumos y consumos que este requerirá.

Hemos manejado ya algunas funciones: **scanf()**, **printf()**, **gets()**, **abs()**, etc. Cualquier función en C posee la misma estructura.

II.3.1 Estructura de una función.

En el epígrafe II.1.1 se mencionaba que el lenguaje C es un conjunto de funciones que inician a ejecutarse a través de la función **main()**. De hecho **main()** es una función que realiza una tarea específica.

Una función es un programa que recibe una serie de datos a través de una lista y que devuelve determinada información de un tipo específico. A la lista de datos que la función recibe se le conoce con el nombre de lista de parámetros.

Tal y como se había mencionado la forma general de un programa C se muestra en la siguiente figura:

```
declaraciones globales
tipo-devuelto main(lista de parámetros)
declaraciones de parámetros;
{
      secuencia de sentencias
}

tipo-devuelto nombre-de-funcion_1(lista de parámetros)
declaracion de parámetros;
{
      secuencia de sentencias
}
.
.
.
tipo-devuelto nombre-de-funcion_n(lista de parámetros)
declaracion de parámetros
{
      secuencia de sentencias
}
```

Cada función debe poseer un nombre específico. Es con ese nombre que un programa puede realizar una llamada a dicha función. El tipo devuelto indica al compilador cual debe ser el tipo de dato devuelto por la función como resultado al programa que realiza la función. Si el programa no devuelve ningún tipo de valor se antepone el especificador **void**. C tal y como está especificado por Kernigal & Ritchie no requiere de especificar el tipo de dato devuelto. C ++ exije que esto sea especificado.

Iniciaremos el bloque de ejemplos de las funciones con un programa sencillo que realiza dos llamadas. El programa `main()` utilizará las funciones `hola()` y `adios()` para desplegar un cortés saludo al usuario y despues despedirse de él.

Ejemplo:
Programa que muestra la forma de realizar un programa con funciones sencillas.

```
/* Ejemplo 30 */
/* 23/Nov/92 */
/* Ejemplo: Llamada a funciones */

main()
{
      clrscr();
      printf("\n Este es el programa principal");
      printf("\n Realiza la llamada a la funcion HOLA");
      hola();
      printf("\n\n Realiza la llamada a la funcion ADIOS");
      adios();
      printf("\n\n Fin de programa.\n En el programa principal");

}
hola()
{
      printf("\n\n Esta es la funcion hola");
      printf("\n Hola amigos");
}
adios()
{
      printf("\n\n Esta es la funcion adios ");
      printf("\n  ¡ Adios amigos!");
}
```

Resultados:

```
Este es el programa principal
Realiza la llamada a la funcion HOLA

Esta es la funcion hola
Hola amigos

Realiza la llamada a la funcion ADIOS

Esta es la funcion adios
 ¡ Adios amigos!

Fin de programa.
En el programa principal
```

Obsevaciones:

Las funciones son declaradas utilizando solamente su nombre. Y la llamada que el programa main() hace, utiliza únicamente dicho nombre para indicar que desea la ejecución de la función. Tan pronto como una función termina su ejecución, devuelve el control al programa que realiza la llamada.

Ejemplo:
Escribir una función que eleve un número entero a cualquier potencia entera

Solución:

```
/* Ejemplo 32 */
/* 23/Nov/92 */
/* Ejemplo: Cálculo de potencias enteras */

/* Archivo de encabezado usado para definir el máximo entero   */
#include <limits.h>

/* Si su compilador no lo acepta puede definir al máximo entero de
la siguiente manera:

const int MAX_INT = 32767;

Recuerde que el máximo número depende de la implantacion que trabaje */

main()
{
   int base, potencia;
   int resultado;
   char opcion;
   const int ESC=27;



   /* Ciclo de control del programa principal */
   opcion = '';
   while (opcion != ESC )
   {
      clrscr();
      printf("\n\tCálculo de potencias");
      printf("\n\t====================\n\n");

      /* Lectura de los datos */
      do
      {
            printf ("\n Base     : ");
            scanf("%d",&base);
      } while (base == 0);
      do
      {
            printf( "\n Potencia : ");
            scanf("%d",&potencia);
      } while (potencia <= 0);
```

```
        /* Llamada a la funcion
        resultado = poten(base,potencia);

        /* Despliege de resultados     */
        if (resultado != 0)
           printf("\n %d elevado a la %d = %d", base,potencia, resultado);
        else
           printf("\n Error de desborde");

        printf("\n\n Pulse una tecla para continuar... <ESC>: SALIDA");
        opcion = getch();


    }
}

/* Esta funcion eleva un entero a cualquier potencia entera
 su forma general es

        poten(base, potencia);

        el resultado es un número entero.
        Si la función detecta desborde (overflow) devolver  0
        como código de error.
 */
poten (numero, potenci)
int numero, potenci;
{
        int indice,resultado;
        resultado = 1;

        for (indice = 1; indice <= potenci && resultado != 0 ; indice++)

                /* SI (no se detecta error de desborde)  */
                if (INT_MAX/numero >= resultado)
                    /* Realiza el cálculo     */
                    resultado = resultado*numero;
                else

                    /* Asigna código de error      */
                    resultado = 0;

        return (resultado);
}
```

Observaciones:

El programa hace uso del archivo de encabezado **limits.h** en donde se encuentra la definición de la constante MAX_INT. Dicha constante define cual es el máximo entero disponible en el sistema. Este valor es utilizado para detectar una condición de desborde en la variable que almacenará el resultado de la potenciación.

El archivo de encabezado se encuentra bajo el subdirectorio *include* del directorio en donde se

encuentre su compilador. Si el archivo no se encuentra disponible, puede sustituir directamente la declaración por la línea **const int MAX_INT = máximo número entero disponible en su sistema.** Es importante observar que dicha declaración se está colocando fuera de la función `main()`. En el epígrafe II.3.3 se explicará la razón de ello.

Así como la lista de parámetros sirve para recibir datos desde el programa que realiza la llamada. La sentencia `return` especifica cual es la variable que retomará su valor hacia el programa que realizó la llamada. Dicho valor se condidera como resultado de la función. A menos que se especifique otra cosa, el valor que se espera como resultado de una función debe ser entero. Este caso se discutirá con detalle en párrafos posteriores.

La condición de error se detecta antes de que este suceda para poder tomar una acción preventiva y poder retener el control de la ejecución del programa. Si un error es de tal magnitud que el programa aborta la ejecución y el sistema operativo retoma el control de los procesos, seguramente se tendrá por resultado un usuario sumamente molesto que habrá perdido información y tendrá que volver a realizar todo el proceso para entrar a su aplicación y recapturar sus datos.

La función no solo devuelve un resultado correcto. En caso de detectar una condición de error, tal como el desborde de una variable, entrega al programa que ha realizado la llamada un código de error que le indica que ha sucedido una situación inesperada. La función no produce un despliege en la pantalla para indicar tal situación de error. La rutina no tiene por función desplegar mensajes sino realizar cálculos.

Imaginemos que deseamos resolver un programa en donde sea condición del problema encontrar un número que no provoque desborde en el cálculo del factorial. El proceso de búsqueda de tal número debe ser transparente al usuario, es decir, el usuario no se debe dar cuenta de que existe tal proceso. Si se escribe la función para que despliege el mensaje de error, ya no podremos utilizarla en dicha aplicación y será menester escribir otra función que realiza la misma tarea pero sin desplegar mensajes, lo cual es a todas luces algo totalmente ineficiente y que provocará confusión en las personas que lean nuestro programa. Recordemos que al escribir funciones, debemos pensar en que pueden ser utilizadas por otros programas que no necesariamente sean semejantes al que se está resolviendo. Esto nos permitirá escribir un código más general y por tanto reutilizable.

En esta función se presenta una característica muy importante en las funciones: El paso de parámetros. En el capítulo uno se mencionaba que estos parámetros deben está claramente definidos en la lista de parámetros de la función.

Desde el programa principal los datos son colocados en la lista de parámetros de la función. Obsérvese que los nombres de las variables que aparecen en la lista son `base` y `potencia`. La función en cambio recibe los valores en las variables `numero` y `potenci`. Durante el proceso de llamada a la función se realiza una copia de los valores que están colocados en la lista de parámetros de la llamada a las variables que están colocados en la lista de parámetros de la función. Dado que se realiza una copia de valores no son trascendentes los nombres de las variables en la llamada, pero si su tipo.

base — es copiado en ——> numero
potencia — es copiado en ——> potenci

A esta forma de pasar los parámetros se le conoce como *paso de parámetros por valor* ya que solo el valor de las variables del programa que realiza la llamada son transferidos a la función. Aunque en la función cambien los valores de las variables, sus contrapartes en el programa de llamada no son afectados.

Programa:
Escribir una función que calcule el factorial de un número.

Solucion:
El factorial de un número *n* que definido por la multiplicación de :

$$n! = n * n\text{-}1 * n\text{-}2 * n\text{-}3 * ... * 3 * 2 * 1$$

Para realizar el programa se debe tomar el número n e irlo multiplicando por el número n-1 que le antecede. Tan pronto como esto ocurre es necesario preguntar si ya se llegó al número 1 ya que si no es así se debe decrementar el número y repetir todo el proceso.

```
/* Ejemplo 33 */
/* 25/Nov/92 */
/* Ejemplo: Cálculo del factorial. */

/* Linea usada para definir el máximo entero   */
#include <limits.h>

/* Si su compilador no lo acepta puede definir al máximo entero de
la siguiente manera:

const int LONG_MAX = 2147483647;

Recuerde que el máximo número depende de la implantacion que trabaje */


main()
{
   int numero;
   int resultado;
   char opcion;
   const int ESC=27;



   /* Ciclo de control del programa */
   opcion = '';
   while (opcion != ESC )
   {
      clrscr();
      printf("\n\tCálculo del factorial ");
      printf("\n\t=====================\n\n");

      /* Lectura de los datos */
      do
      {
            printf( "\n Factorial de ?  ");
            scanf("%d",&numero);
            if (numero < 0 )
                  printf(" \n El numero debe ser mayor a cero");
      } while (numero < 0);

      /* Llamada a la funcion  */
      resultado = factorial(numero);
```

```
      /* Despliege de resultados    */
      switch (resultado)
      {
            case  0:
               printf("\n Error de desborde");
               break;
            case -1:
               printf("\n Error en el argumento");
               break;
            default:
               printf("\n El factorial de %d es: %d", numero, resultado);
      }


      printf("\n\n Pulse una tecla para continuar... <ESC>: SALIDA");
      opcion = getch();


   }
}

/* Esta funcion obtiene el factorial de un número entero
   Su forma general es:


            factorial (numero);


   El resultado es un número entero largo.
   Los códigos de error devueltos son:
       0    error de desborde (overflow)
      -1    error en el argumento
 */

factorial (numero)
int numero;
{
      long int resultado;
      resultado = 1;
      if (numero >= 0)
          while (numero > 1   &&   resultado != 0)
          {
               if (LONG_MAX/numero >= resultado)
               {
                  resultado = resultado*numero;
                  numero--;
               }
               else
                  resultado = 0;
          }
      else
          resultado = -1;

      return (resultado);
}
```

Observaciones:

Obsevese que tanto en `main()` como en `factorial()` existe una variable llamada `numero`. El valor de la variable `main -> numero` es transferido a `factorial()`. Una vez en dicha función el valor es decrementado sucesivamente hasta alcanzar el valor de 1. El último valor de `factorial -> numero` es por tanto 1. Pero en el programa que ha realizado la llamada la variable no ha sufrido ningún cambio. Aunque ambas variables tienen el mismo nombre simbólico, en realidad ocupan direcciones diferentes de memoria.

Las ventajas que esto acarrea son múltiples, por un lado nos permiten diseñar funciones en forma totalmente **independiente** a cualquier otra existente en el sistema. No es necesario saber si existe en alguna parte del sistema una variable que se llama igual que la que vamos a manejar en el módulo que estamos escribiendo, Si esto ocurre podemos tener la certeza de que serán manejadas con total independencia.

Un lector avispado ya habrá notado que existe redundancia en la validación de los datos. Por una parte en la lectura del número al cual se le calculará su factorial, se realiza una primerá validación para evitar que se introduzca un número al cual no se le pueda calcular su factorial, como podría ser un número negativo.

Por otra parte, la función `factorial()` también realiza una validación del argumento que le ha sido pasado con el fin de asegurarse de que dicho argumento es válido. ¿ Porque se ha escrito el programa de esta manera?.

Continuamos siguiendo la filosofía de la escritura de programas generales y reutilizables. La función `factorial()` es una tarea que intuitivamente bastará con escribir una sola vez y después la utilizaremos en inumerables programas, cada uno de los cuales será distinto de los otros. Dado que no sabemos cuales serán las condiciones que prevalecerán en cada uno de esos *Inumerables* programas, debemos escribir una rutina que suponga que los datos que se le envian son inválidos. La función debe checar los parámetros por si misma y devolver un código de error si algo erróneo sucede.

El programa que realiza la llamada no puede espera. que sea la función quien realice la validación de los datos, ya que de ser así esto sucedería hasta que nos encontremos en el segmento de cálculo de información. Detectar un error hasta este punto implicaría un gran esfuerzo por parte del programador para regresar al segmento que realiza la lectura del dato inválido o , en el peor de los casos, la relectura de todo el bloque de datos necesarios. Esto no será agradable para ningún usuario. Ya que dicha detección de un dato erróneo por parte de la rutina `factorial()` podría ocurrir justo un momento antes de concluir un proceso que ha tardado 32 horas de proceso.

En la medida de lo posible, todo bloque de lectura de datos debe validar la información de entrada. A su vez en el momento de la escritura de una función o un programa, se debe partir inicialmente de la suposición de que los datos son válidos. Tan pronto como dicho código esté correcto, se le debe recubrir con una capa de código que se asegure de que eso es cierto. Dicha técnica de programación es especialmente útil cuando se trabaja sobre tiempo para la entrega de los programas.

Se ha mencionado que el lenguaje C es una colección de funciones. La función `main()` no es un excepción y así como es capaz de enviar un argumento a cualquier función a la que llame, también puede recibir argumentos del programa que lo llama. Para poder ejecutar cualquier programa, sea en el equipo que sea, primero es necesario realizar una llamada desde el sistema operativo. El programa superordinado de la función `main()` es el sistema operativo. Desde el sistema operativo es posible mandar una lista de parámetros a la función `main()` para que este trabaje.

El paso de parámetros se realiza en la misma línea de comandos del sistema operativo. Un programa

que se parametriza de esta manera tiene una apariencia mucho más profesional. Por ejemplo, un programa que coloque las sangrías correctas en un programa mal sangrado lucirá mucho mejor si lee el nombre del archivo desde la línea de comandos.

La función `main()` recibe los argumentos de la línea de comandos a través de dos argumentos predefinidos en el lenguaje:

- **argc** devuelve un número entero en el cual se indica cuantos argumentos han sido escritos en la línea de comandos
- **argv** es un puntero a un arreglo de tipo caractéres en el cual cada elemento apunta a un argumento suministrado por la línea de ordenes. Es decir, el prompt de sistema operativo.

De esta forma es posible saber cual es el número de argumentos sumistrados al programa y evitar un desborde al intentar accesar datos no existentes en el arreglo argv. El primer argumento siempre será el nombre del programa, de donde se colige que argc devuelve siempre como mínimo 1. En la práctica, el número de argumentos está limitado por la capacidad del buffer del sistema operativo, esto significa que solo es posible utilizar una pequeña cantidad de argumentos de entrada. En teoría es posible utilizar hasta 32767 argumentos.

Para ilustrar los conceptos vertidos veamos el siguiente :

Programa:
Hacer un programa que dibuje en la pantalla el triángulo mostrado . El número de líneas debe leerse desde la línea de comandos.

```
    1
   212
  32123
 4321234
543212345
```

. . .

Solución:
En un ejemplo anterior se había dibujado la parte derecha del triángulo. Ahora solo es necesario implantar la parte izquierda. Así como en el ejemplo anterior todo lo que había que hacer era dibujar una línea de números en orden ascendente. Ahora debemos dibujar un triángulo de espacios en blanco. Un triángulo de números en orden descendente y un triángulo de números ascendentes.

Dicho de otro modo, por cada línea a imprimir es necesario desplegar una serie de caractéres en blanco. Después una serie de numeros en orden descendente para finalmente desplegar una nueva serie de números en orden ascendente.

Pseudocódigo:

```
Dibuja un triángulo
      DECLARACION DE VARIABLES
      SI ( no existen suficientes argumentos) ENTONCES
            ESCRIBE(" Use: trian número-de-líneas")
      SINO
            LEE (número_renglones)
            SI (es un número de renglones válido )
                  LIMPIA PANTALLA
```

```
                HAZ MIENTRAS (no se pinten todos los renglones)
                     PINTA los blancos
                     PINTA cuenta regresiva
                     PINTA cuenta ascendente
                     EJECUTA UN salto_de_linea
                FIN DEL HAZ
           SINO
                ESCRIBE("El número de renglones puede oscilar entre 2-24")
           FIN DEL SI
     FIN DEL SI

/* ejemplo ii.3.2.2
Dibuja un triángulo en la pantalla
Hace uso de envio de parámetros desde la línea de comandos */


void main(int argc, char *argv[])

{
     /* Declaracion de variables */
     int con_ren, con_col, limite;
     int contador,con_blancos;
     int centro = 40;
     char resp;


     if (argc < 2)
        /* Se indica cual es la forma correcta de llamar al programa
           Aqui suponemos que el programa recibe el nombre de TRIAN
           para el entorno DOS.
           En entorno Unix el programa recibe el nombre de a.out */
        printf("\n Use: \n trian número-de-líneas");
     else
     {

        /* lectura del número de renglones a pintar */
        limite = atoi(argv[1]);

        /* Si es un número de renglones válido */
        if (limite > 1 && limite < 24)
        {

          /*Limpia pantalla y dibuja el triángulo */
            clrscr();
          con_ren=0;

          while (con_ren < limite)
          {
           ++con_ren;
           con_col=0;
```

```
            /* Pinta los blancos */
            contador= centro-con_ren;
            if (con_ren > 9 )
               contador = contador - (con_ren - 9 );
               con_blancos=1;

            while (con_blancos < contador)
            {
                  printf(" ");
                  con_blancos++;
            }

            /* Pinta cuenta regresiva */
            contador = con_ren;
            while ( contador > 1)
            {
                  printf("%d",contador);
                  contador--;
            }
            con_col=1;

            /* Pinta cuenta ascendente */
            while (con_col <= con_ren)
            {
                  printf("%1d", con_col);
                  con_col++;
            }
            printf("\n");
            }

         }
         else
           printf ("\nEl número de renglones puede oscilar entre 2-24\n");
      }

}
```

Resultados:

```
C:\COMP\TC2>trian

 Use:
 trian numero-de-lineas
```

```
C:\COMP\TC2>trian 11
                                    1
                                   212
                                  32123
                                 4321234
                                543212345
                               65432123456
                              7654321234567
                             876543212345678
                            98765432123456789
                           1098765432123456789 10
                          111098765432123456789 1011

C:\COMP\TC2>
```

Obsevaciones:

En general la declaración de los parámetros de la función **main()** se hace tal y como se mostrado en el ejemplo. El argumento argc se declara como entero y el argumento argv como un puntero a un arreglo. Dentro de los corchetes del arreglo no se ha puesto ningún número. Esto es necesario ya que aunque sepamos que el programa solo requerirá de dos argumentos nosotros no estamos seguros de que el usuario solo digitará dos argumentos. Los corchetes vacíos indican que existe un arreglo de una longitud indeterminada.

La variable **centro** indica que en la columna 40 se tiene el centro de la pantalla desde la cual se dibujará el eje del triángulo. La primer validación que realiza el programa es referente al número de argumentos ya que si no se poseen los argumentos necesarios el programa no podrá realizar su función.

El número de renglones que se desean para dibujar el triángulo es tomado desde la línea de comandos y colocado en el elemento **argv[1]** ya que **argv[0]** tiene almacenado el nombre del programa. Ambos elementos son de tipo caractér y no es posible utilizar directamente un caractér para operaciones numéricas. Para ello es necesario realizar la conversión de caractér a numérico a través de la función **atoi()**.

Programa:
Realizar un programa que haga converger un letrero pasado como argumento desde la línea de comando, en una línea especificada por el mismo medio.

Solución:
Para la ejecución de este programa hacemos uso de la funcion **gotoxy()** la cual posiciona el cursor en las coordenadas definidas en los argumentos de dicha función. En el apéndice C de estos apuntes se muestra una forma de implantar la función **gotoxy()** si su compilador no la soporta pero su terminal es compatible con ANSI.

```
/* Econverg.c
27 de nov de 1992
Este ejemplo muestra como utilizar los argumentos de la función main()
Un letrero introducido desde la línea de comandos es desplegado*/

#include <stdio.h>
#include <conio.h>     // Prototipo de gotoxy()
#include <string.h>
void converge (int linea, char *mensaje);
```

```
int main(int argc, char *argv[])
{
      int linea,coderr;

      if (argc > 2)
      {

            clrscr();
            linea=atoi( argv[1] );
            if (linea > 0 && linea < 25)
            {
                  converge(linea,argv[2]);
                  printf("\n");
                  coderr=0;
            }
            else
            {
                  printf("\nEl numero de linea debe ser positivo y menor a 25\n");
                  coderr=1;
            }
      }
      else
      {

            printf("\nUse: econverg linea mensaje \n");
            coderr=1;
      }
      return(coderr);
}


void  converge(int linea, char *mensaje)
{
      int i,j,espera,inicio;
      inicio=40-strlen(mensaje)/2;
      for (i=1, j=strlen(mensaje); i<=j; i++,j--)
      {
            for (espera=1; espera < 9000; espera++);
            gotoxy(i+inicio,linea);printf("%c",mensaje[i-1]);
            gotoxy(j+inicio,linea);printf("%c",mensaje[j-1]);
      }
}
```

Resultados:

```
C:\COMP\TC2>econverge

Use: econverg linea mensaje
```

```
C:\COMP\TC2>econverge 11 Hola.alumnos.de.lenguaje.C
```

```
                    Hola.alumnos.de.lenguaje.C

C:\COMP\TC2>
```

Observaciones:

El programa utiliza tres archivos de cabecera necesarios para su correcta compilación.

Además este programa no tiene al declarador de tipo `void` en la línea de `main()`, en su lugar encontramos el declarador `int` que indica que la función `main()` devolverá al sistema operativo un número entero al terminar su ejecución.

La variable entera `coderr` funge como una bandera que recoge un código de error que será devuelto al sistema operativo. Dicho código de error podrá ser útil cuando este programa (o cualquier programa) sea ejecutado por un comando de procesamiento en lotes. Si el programa termina sin ningún problema devolverá un código de error igual a cero. Si por el contrario es detectada una condición anormal de funcionamiemto se devuelve un código igual a 1. Observense las técnicas usadas para mantener al programa sin el uso de saltos incondicionales aún cuando sea detectado un error irrecuperable. En todo momento el programa se mantiene como un programa proplo.(inicio por arriba, lectura de arriba a abajo y salida por abajo)

En el caso de los sistemas bajo el sistema operativo DOS, un archivo **bat** podrá detectar dicho código y tomar las medidas pertinentes. En los sistemas bajo plataforma UNIX un shell ocupará el lugar correspondiente en la decisión de las acciones a tomar. En seguida se muestra un ejemplo para DOS de un archivo bat para la detección de dicho código de error. El archivo se llama conv.bat y se activa desde la línea de comandos exactamente igual que se usara con el programa **econverg.exe**

```
@echo off
echo.
echo.
echo          Programa detector del código de error
echo.
echo.
econverge %1 %2
if errorlevel 1 goto error
goto fin
:error
echo  Se detecto error
:fin
echo Hecho...
```

II.3.2 Funciones prototipo.

Se ha mencionado que las funciones en C devuelven un dato que es de determinado tipo y reciben datos de un determinado tipo. En general es muy recomendable especificar el tipo de datos que las funciones devolverán al programa que los llame. Esto nos permitirá conocer desde el tiempo de compilación cuando existan errores en los tipos entregados y esperados en los programas. Si los tipos no coinciden y no son declarados explícitamente C podría compilar exitósamente pero ejecutar erróneamente. C al igual que muchos lenguajes procesarán la información que reciban aunque está produzca un resultado erróneo o definitivamente absurdo. Recordemos el clásico dicho en computación: *Si basura entra, basura sale.*

Una **función prototipo** es una declaración de tipo para una función utilizada en un programa. Dicha declaración de tipo se realiza en la función con un fin semejante a la declaración de tipo de una variable normal.

La declaración de tipo se realiza en dos partes distintas del programa: antes de la función **`main()`** y en la declaración de la función. Si la función no devuelve ningún parámetro (como puede ser el caso de la función **`main()`**) o si no requiere de alguno, se hace uso de especificador de tipo **void**. Recuerde que C++ exige la declaración del prototipo de las funciones.

La forma general de un prototipo de función es el siguiente:

tipo nombre_de_funcion(lista de parámetros)

Por ejemplo para declarar a la función **factorial()** como de tipo long int se debe escribir

long int factorial(numero)

Ahora bien, existen dos formas de realizar la declaración de los parámetros de la función. Hasta este momento hemos utilizado la forma:

```
nombre_de _funcion (lista de parámetros)
declaración de parámetros
{
        bloque de instrucciones
}
```

Esta forma es conocida como la forma clásica definida por Ritchie & Kernigal. El estandar ANSII define una forma alterna para la declaración que es conocida como la *forma moderna*. La forma moderna es definida de la siguiente manera:

```
tipo nombre_de_funcion (tipo argumento_1, tipo argumento_2, tipo argumento_3,...,tipo argumento_n)
{
        bloque de instrucciones
}
```

Es decir, ya no se declaran los argumentos en una línea aparte sino en la misma lista de argumentos.

Programa:
En el epígrafe anterior el programa que calcula el factorial recibe datos de tipo entero y devuelve un resultado entero. La rutina **`factorial()`** no nos es de gran utilidad si se le define así, ya que el mayor factorial que puede producir sin que se produzca un error, no es mayor a 8 ó 10. Reescribir el programa para que devuelva un dato del tipo long int.

```
/* Ejemplo 34 */
/* 25/Nov/92 */
/* Ejemplo: Cálculo del factorial.
      El resultado de la funcion es long int */

/* Línea usada para definir el máximo entero    */
#include <limits.h>

/* Si su compilador no lo acepta puede definir al máximo entero de
la siguiente manera:

const int LONG_MAX = 2147483647;

Recuerde que el máximo número depende de la implantacion que trabaje */

/* Prototipo de la funcion */
long int factorial ( long int numero);

void main(void)
{
   long int numero;
   long int resultado;
   char opcion;
   const int ESC=27;

   /* Ciclo de control del programa */
   opcion = '';
   while (opcion != ESC )
   {
      clrscr();
      printf("\n\tCálculo del factorial ");
      printf("\n\t=====================\n\n");

      /* Lectura de los datos */
      do
      {
            printf( "\n Factorial de ?  ");
            scanf("%ld",&numero);
            if (numero < 0 )
                  printf(" \n El numero debe ser mayor a cero");
      } while (numero < 0);

      /* Llamada a la funcion  */
      resultado = factorial(numero);

      /* Despliege de resultados     */
      switch (resultado)
      {
            case  0:
               printf("\n Error de desborde");
               break;
```

```
            case -1:
               printf("\n Error en el argumento");
               break;
            default:
               printf("\n El factorial de %ld es: %ld", numero, resultado);
      }

      printf("\n\n Pulse una tecla para continuar... <ESC>: SALIDA");
      opcion = getch();
   }
}

/* Esta funcion obtiene el factorial de un numero entero
   Su forma general es:


            factorial (numero);


   El resultado es un número entero largo.
   Los códigos de error devueltos son:
       0     error de desborde (overflow)
      -1     error en el argumento
 */

long int factorial (long int numero)
{
      long int resultado;
      resultado = 1l;
      if (numero > 0l)

          while (numero > 1    &&    resultado != 0l)
          {
               if (LONG_MAX/numero >= resultado)
               {
                  resultado = resultado*numero;
                  numero--;
               }
               else
                  resultado = 0;
          }
      else
          resultado = -1;

      return (resultado);
}
```

Observaciones:

La funcion main no devuleve ni acepta parámetros. Por eso se le ha agregado a su declaración el especificador void.

Como tarea mental queda al lector pensar si es realmente útil declarar a **factorial()** como long int o si es necesario declarar otro tipo de dato para el valor devuelto.

Un error sutil en la programación utilizando la forma clásica de declaración de parámetros de una función prototipo es escribir las siguientes líneas:

```
/* Prototipo de la funcion */
long int factorial ( long int numero);
.
.
.
/* declaracion de la función */
factorial (numero)
long int numero
{
      long int resultado;
      .
      .
      .

      return (resultado);
}
```

En este segmento de código se declara la función prototipo para que **factorial()** pueda esperar y devolver un dato del tipo long int. La codificación de la función **factorial()** indica que se recibirá un dato del tipo long int y declara que la variable **resultado** es a su vez de tipo long int . Cabría esperar por tanto que el compilador *comprendiera* que la función **factorial()** en efecto devolverá un dato long int. Esto no ocurre. Dado que en la declaración de la función no se indicó el tipo que se espera de **factorial** el compilador toma por omisión el tipo entero. La variable **resultado** es convertida de long int a int sencillo y el compilador detecta un error en la confrontación de tipos. La forma correcta de realizar la declaración es por tanto la que se ha mostrado en el programa:

```
/* Prototipo de la funcion */
long int factorial ( long int numero);
.
.
.
/* declaracion de la función */
long int factorial (numero)
long int numero
{
      long int resultado;
      .
      .
      .

      return (resultado);
}
```

II.3.3 Otros tipos de variables

En general, los tipos básicos son suficientes para la gran mayoría de las aplicaciones. Sin embargo, por razones de claridad o de portabilidad es deseable que el programador pueda redefinir tipos de datos.

La sentencia typedef permite redefinir un tipo asociandole un nuevo nombre. Esto significa que a determinido tipo de dato se le puede declarar utilizando un nombre más descriptivo. El hecho de que asocie un nuevo nombre a un tipo de dato específico no significa que se cree un nuevo tipo de dato. Simplemente se define una nueva manera de llamarlo.

La forma general para definir tipos de datos es:

typedef tipo identificador;

Veamos el siguiente ejemplo:

```
/* Definicion de nuevos tipo de datos */
#include <conio.h>

main()
{

      /* Definicion del tipo LOGICO y de dos constantes*/
      typedef int logico;
      const logico verdadero=1;
      const logico falso=0;

      logico continua=verdadero;

      /* Ahora se define el tipo BANDERA*/
      typedef logico bandera;
      bandera se_acabo=falso;

      /* El tipo INT aun se puede usar  */
      int contador= 0;
      char tecla;


      while (continua)
      {
            contador++;
            printf("\nValor del contador: %2d", contador);

            if (contador >2)
                  se_acabo=verdadero;
            else
            {
                  printf("\n<Esc> para finalizar");
                  tecla = getch();
            }
```

```
            if (tecla == 27 || se_acabo)

                continua=falso;

        }
}
```

Resultados:

```
Valor del contador:  1
<Esc> para finalizar <esc>

Valor del contador:  1
<Esc> para finalizar <enter>
Valor del contador:  2
<Esc> para finalizar <enter>
Valor del contador:  3
```

Observaciones:

Hasta este momento, hemos definido a las variables lógicas por medio del tipo int. Para diferenciar una variable lógica de una variable numérica, solo disponíamos de la opción de un comentario para aclarar el significado de la variable, o bien, teníamos la esperanza de que el nombre de la variable sugiriera su contenido.

En el programa presentado, definimos un tipo `logico` con el cual declararemos todos los datos lógicos que sean requeridos en el programa. De esta forma la naturaleza de las variables quedará claramente expresada. Los dos valores lógicos de nuestro nuevo tipo de dato son definidos como constantes porque no deseamos que sean modificados a lo largo del programa.

En base al tipo `logico` que hemos definido vamos más allá al definir un tipo de dato extra. Una bandera es un caso particular de una variable lógica, así que definimos el tipo `bandera` para hacer clara la naturaleza de las variables que tengan ese uso dentro de nuestro programa.

El tipo `int` no ha sido anulado ni redefinido, solo se le han asociado nuevos nombre con los cuales se pueden declarar variables, por ello es posible definir datos de tipo entero después de las declaraciones con `typedef`. En ninguno de los casos se ha creado un nuevo tipo de dato. Tanto las variables del tipo `logico` como las del tipo `bandera` son enteros.

II.3.4 Clases de almacenamiento

Cuando se declara una variable, el compilador asume en forma automática el tipo de almacenamiento que tendrá dicha variable dentro de la memoria. En general se asume que las variables son asignadas a una localidad de memoria cuando la función es accesada y que dicha variable desaparece cuando la función devuelve el control al programa padre que la llamo.

El lenguaje C permite especificar otros tipo de almcenamiento. El especificador de almacenamiento precede al resto de la declaración de variable. La forma general es :

especificador_de_almacenamiento tipo lista_de_identificadores;

Donde : *Especificador_de_almacemaniento* es uno los cuatro siguientes:

extern
static
register
auto

EL ESPECIFICADOR EXTERN

Una de las ventajas de construir y diseñar los programas en forma modular estriba en que se pueden asignar a distintos programadores el trabajo de programación para unirse despúes en la fase conocida como *integración de sistemas*. Además, un mismo programador puede fraccionar su trabajo en módulos que puede programar en archivos distintos y compilar en forma separada para acelerar los tiempos de compilación.

El compilador del lenguaje C nos permite enlazar distintos módulos de un programa colocados en archivos distintos (Vease epígrafe II.3.8). Y las variables declaradas en forma global las puede conservar a través de los distintos archivos en que se fragmente el programa.

Sabemos que una variable global solo puede ser declarada una sola vez a lo largo del programa. Dicha variable no necesitará ser declarada en cada una de las funciones del programa, y cada una de ellas tendrá acceso a dicha variable. El ámbito de visibilidad es general. Si se intenta declarar más de una vez con el mismo nombre, el compilador indicará un error al no saber que hacer con la duplicidad de las variables ya que no posee medios para saber cuando usar una y cuando usar otra.

Cuando se utilizan programas repartidos en distintos archivos el compilador detectará como un error si encuentra que una variable ha sido declarada una vez en cada uno de los archivos. En tiempo de compilación el error no será desplegado, pero puede saltar a la vista cuando se este enlazando y se intenten crear dos variables distintas con el mismo identificador. Las versiones avanzadas de C (C++ de Borland), pueden interpretar correctamente que se trata de la misma variable y funcionar sin causar conflicto. Por compatibilidad es preferible utilizar el especificador.

Observese el siguiente ejemplo:

Archivo eextem.c

```
#include "ajena.c"
void otra(void);
void ajena(void);

int x;
void main()
{
otra();
ajena();
}
void otra (void)
{
x=100;
}
```

Archivo eajena.c

```
extern int x;
void ajena()
{
printf ("%d",++x);
}
```

En el ejemplo precedente se tienen dos archivos separados. Uno es **eextern.c** el cual contiene a la función **main()**. En dicho archivo se declara como global a la variable x. Dentro de un archivo separado, que en el ejemplo recibe el nombre de **eajena.c** se hace uso de dicha variable en la función **ajena()**. Con el especificador extern se indica que la variable procede de un archivo externo evitando la duplicidad de símbolos. Esto conserva a la variable **x** global a todos los archivos que utilicen la declaración **extern** para definirla.

EL ESPECIFICADOR STATIC

Se ha mencionado que cuando una función devuelve el control a un módulo que la ha mando llamar, las variables asociadas a dicha función desaparecen y sus valores se pierden. Cada vez que la función es llamada, las variables son vueltas a crear y los valores tienen que reinicializarse.

Con el especificador de almacenamiento **static** podemos indicar al compilador que una variable deberá conservar sus valor entre llamadas. Una variable **static** no gana o pierde visibilidad.

Variables estáticas locales

El ámbito de visibilidad de una variable local está restringido a la función que realiza la declaración. Con la añadidura de la especificación *static* la variable continúa teniendo el mismo ámbito de visibilidad, pero ahora no pierde los valores entre llamadas.

Variables estáticas globales

Sabemos que una variable global desaparece entre llamadas a archivos y que si deseamos que sea global a todos los archivos debemos utilizar el especificador extern. El ámbito de visibilidad de una variable global se encuentra definido por el archivo en el cual se encuentre definida dicha variable. Una variable global estática conservará su valor entre llamadas al archivo en el cual se encuentre la variable global.

Ejemplo

VARIABLE REGISTER

Dentro del C definido por Kemigal and Ritchie se definía a una variable **register** como aquella que mantenía su valor en un registro del microprocesador para acelerar de manera significativa el procesamiento de la información. Los tipos de datos aceptados como registro eran los enteros y los caractér.

El estándar ANSI ha ampliado la definición para que se pueda aplicar la declaración a cualquier variable con el fin de que el acceso sea lo más rápido posible. El compilador entonces manejará las variables para que reciban un trato preferencial. Las variables de tipo entero y caractér aún continuan almacenandose directamente en los registro del microprocesador.

Una variable de tipo **register** entera suele usarse para ciclos controlados por variable. Al ser almacenada en el microprocesador los incrementos en el valor de la variable de control se efectúan con mayor velocidad aumentando la eficiencia del programa.

En los que concierne a otros tipos de variables, el estándar ANSI especifica que el compilador puede llegar a ignorar la especificación `register`, aunque esto rara vez ocurre en la práctica.

Ejemplo:
La función `potenci()` que se escribió anteriormente para elevar un número a una potencia entera puede reescribirse de la siguiente manera:

```
/* Esta funcion eleva un entero a cualquier potencia entera
 su forma general es

      poten(base, potencia);

      el resultado es un número entero.
      Si la función detecta desborde (overflow) devuelve  0
      como código de error.
 */
poten (numero, potenci)
int numero, potenci;
{
      register int indice,resultado;
      resultado = 1;

      for (indice = 1; indice <= potenci && resultado != 0 ; indice++)

            /* SI (no se detecta error de desborde)   */
            if (INT_MAX/numero >= resultado)
                /* Realiza el cálculo     */
                resultado = resultado*numero;
            else

                /* Asigna código de error      */
                resultado = 0;

      return (resultado);
}
```

Algo muy importante que hay que hacer notar es que una variable de tipo register no tiene dirección asociada. Debido a que una variable de este tipo se aloja directamente en el microprocesador no tiene una dirección en la memoria RAM dedicada a las variables. Esto significa que no es posible utilizar el operador & a una variable register.

Es posible declarar cualquier cantidad de variables de tipo registro dentro de un programa. Pero como el microprocesador tiene una cantidad limitada de registros internos, solo una pequeña fracción de esas variables serán asignadas realmente a los registros del microprocesador. El resto de las variables serán convertidas a variables de tipo normal. Esto se hace por compatibilidad con otro tipo de procesadores.

EL ESPECIFICADOR AUTO

El especificador auto es definido aún en algunas versiones de C. Hace referencia a variables de tipo automático. Sin embargo la gran mayoría de las implantaciones ya no utilizan este especificador ya que sirve para declarar variables de tipo local.

II.3.5 Estructura general de un programa C

Se ha mencionado que un programa en lenguaje C es una colección de funciones cuya estructura general es la siguiente:

```
declaraciones globales
      tipo-devuelto main(lista de parámetros)
      declaraciones de parámetros;
      {
            secuencia de sentencias
      }

      tipo-devuelto nombre-de-funcion_1(lista de parámetros)
      declaracion de parámetros;
      {
            secuencia de sentencias
      }
      .
      .
      .
      tipo-devuelto nombre-de-funcion_n(lista de parámetros)
      declaracion de parámetros
      {
            secuencia de sentencias
      }
```

Dentro de las **declaraciones globales** se puede declarar algo más que las variables globales o las contantes que se utilizarán a lo largo del programa. Dicho segmento nos permite manipular el programa en tiempo de compilación. Esto significa que el resultado de la operaciones que se realicen aquí no afectarán al programa en plena ejecución sino antes de que éste entre en ejecución.

Básicamente la manipulación del compilador se refiere a la capacidad para el manejo de macros, las opciones del preprocesador de C y la compilación multiarchivo.

Antes de que un archivo fuente entre al compilador de C, dicho archivo es sometido a un procesamiento previo por parte de un *preprocesador* que se encarga de establecer algunas opciones que se discutirán en el siguiente apartado.

La estructura general de un programa en C se puede ahora representar de la siguiente manera:

1.- Los archivos inician con comentarios que describen el propósito del módulo y proveen al programador de alguna otra información que sea de interés para la cabal comprensión. Es muy recomendable que se indique información tal como el nombre del programador, la fecha de creación del programa y la fecha de la última actualización.

2.- Los comandos correspondientes al preprocesdor de C. Estos comandos, como se explicará en el siguiente apartado, reciben el nombre de *directivas del preprocesador.*

3.- El siguiente bloque corresponde a la declaración de variables y funciones que serán visibles en forma global por los programas que conforman el archivo de código fuente.

4.- Enseguida encontramos la función principal de ese archivo, dentro de la cual se definen las variables locales que la función requiera, así como las llamadas a las funciones secundarias.

5.- Por último encontramos las funciones secundarias, con su repectiva declaraciónde variables locales, su cuerpo de proceso y sus llamadas a otras funciones.

II.3.6 Macros

Una macro es la definición de un símbolo que será igualado a algún tipo de segmento de código válido en lenguaje C. De esta forma se puede utilizar el símbolo sin necesidad de escribir el código respectivo en el programa.

Cuando el preprocedaor del lenguaje C accesa el archivo fuente, toma la lista de definiciones de macro y reemplaza cada ocurrencia que exista en el código fuente con el código equivalente.

El uso más común de una macro es la definición de nombre para constantes numéricas. Esto aumenta la legibilidad con los beneficios enunciados en el capítulo uno. En ejemplos pasados utilizabamos la variable ESC para almacenar el valor de la tecla <ESCAPE> y poder salir del un ciclo. De forma análoga se puede definir

```
#define ESC 27
#define PI 3.1415927
#define NULO '\0'
```

Los símbolos son sustituidos por sus valores dentro del código fuente lo cual significa que para el compilador no existirán los símbolos definidos en los macros sino sus valores. En los compiladores orientados a PC no se podrá modificar dentro del depurador el valor de un macro.

Cuando un identificador definido como una macro aparece dentro de un cadena la sustitución no es llevada a cabo. Por ejemplo, en la definicion de `ESC` no se realiza la sustitución en el siguiente ejemplo:

```
#define ESC 27
printf ("ESC");
```

Una característica ,más poderosa de una macro es la sustitución de código que acepta parámetros. Se define una macro con un nombre de función y los parámetros necesarios para su funcionamiento así como el código asociado a dicha función, cuando la macro es substituida, los parámetros son reemplazados ocurrencia por ocurrencia por el preprocesador.

En el siguiente ejemplo se define la función `cuad()` para obtener el cuadrado de un número entero. Posteriormente se utiliza la macro para asignar el valor a la variable y:

```
#define cuad(x)     ((x)*(x))

.
.
.
y=cuad(z)
```

El preprocesador realizará la siguiente substitutición, que será lo que realmente será compilado.

```
y=((y)*(y))
```

Debe tenerse mucho cuidado cuando se utilicen macros con sustitución de parámetros. Debido a que la sustitución ocurre aún antes del tiempo de compilación los parámetros no tendrán un valor cuando sean reemplazados, como ocurre en una función definida con los medios discutidos anteriormente.

Supongase que la función `cuad()` definida como una macro se aplica de la siguiente manera:

```
#define cuad(x) x*x
.
.
.
y=cuad(a+b)/10
```

El preprocesador entregaría el siguiente código:

```
y=a+b*a+b/10
```

Lo cual es claramente incorrecto. Es indispensable cuidar detalles de este tipo porque pueden hacer que el programa no realice lo que el programador desea y arrojando resultados incorrectos los paréntesis aseguran la evaluación en el orden de precedencia correcto. La primera definición de la macro (#define cuad(x) ((x)*(x))) provocaría que el preprocesador entregara el siguiente código:

```
y=((a+b)*(a+b))/10
```

Dicho código es correcto porque será evaluado en el orden de precedencia esperado por el programador.

Una característica nueva del C ANSI es el paso de símbolos (token) a través del operador **##**. Si nosotros tenemos un programa en donde se sustituye un argumento tal y como ha sido mostrado en los ejemplos precedentes y este argumento genera un símbolo nuevo válido, dicho símbolo será a su vez reemplazado por su valor.

Sea la siguiente serie de definiciones:

```
#define MENU1 "Altas"
#define MENU2 "Bajas"
#define Menu(n) MENU##n
```

Cuando el preprocesador encuentre una línea con una macro como `Menu(2)` se realizará la sustitución indicada generando el código `MENU##1`, el cual se transforma por medio del operador ## en `MENU2` que es una definicón de macro válida que provocará la sustición final por el valor definido para `MENU2` que es `"Bajas"`

Otro operador definido en el C ANSI es el operador de cadenas **#** el cual coloca una cadena de carecteres añadiendo las comillas necesarias para que el compilador lo acepte. Vease el siguiente ejemplo

```
#define valor(x)  printf(#x" = %d\n",x)
```

Esta línea define una función que imprimirá el valor de una variable entera para poder rastrearla en el proceso de depuración de un programa. Si escribimos a continuacion la línea de código

```
valor(contador);
```

El preprocesador entregará entonces el código mostrado a continuación:

```
printf("contador"" = %d\n",contador);
```

El C de ANSI especifica que las cadenas deben concatenarse cuando se encuentren adyacentes, lo cual significa que el código generado será:

```
printf("contador = %d\n",contador);
```

Que es lo que hubieramos escrito directamente en el código fuente.

C ANSI define una serie de macros predefinidos. Algunos de ellos se refieren al archivo en proceso de compilación . El macro __FILE__ contiene el nombre del archivo fuente actual y __LINE__ contiene el número de linea en proceso.

MACROS PREDEFINIDOS EN C ANSI

MACRO	DEFINICION
__DATE__	Este es una cadena conteniendo la fecha en que se invoca al compilador de C. Su forma general es "MMM DDD YYY". El mes queda definido por sus tres primeras letras del nombre en inglés, Por ejemplo: "Oct 18 1989"
__FILE__	Esto expande una cadena conteniendo el nombre del archivo fuente.
__LINE__	Este es un decimal entero cuyo valor es el número de línea dentro del archivo fuente.
__STDC__	Esta macro expande a el decimal constante 1 para indicar que el compilador es compatible con el ANSI estándar.
__TIME__	Esta cadena obtiene la hora en la cual se inició la compilación del archivo fuente. Su forma general es "HH:MM:SS". Por ejemplo: "14:48:30".

II.3.7 El preprocesador C

En las versiones primigenias de C el preprocesador era un programa aparte del compilador, sin embargo los compiladores modernos realizan su propia llamada al preprocesador. El preprocesador de C es una primera traducción del código fuente en la conversión a código máquina. El preprocesador actúa a través de instrucciones que son llamadas **directivas** y que inician con el caractér #. Cada directiva debe ser escrita en su propia línea sin punto y coma.

La directiva de definición de macros ya ha sido examinada en el epígrafe anterior.

DIRECTIVAS CONDICIONALES

Existen diversas razones por las cuales podríamos desear compilar solamente algunos segmentos de código, o bien preferir compilar un segmento en lugar de otro. La gran variedad de equipos en los que se puede ejecutar una aplicación, las diversas características que un sistema debe poseer para funcionar bajo diferentes plataformas operativas , la flexibilidad que los programadores necesiten para escribir una aplicación pueden ser razones para poder seleccionar código a compilar. En otro caso se pueden activar sentencias **printf()** solo si una macro DEPURANDO a sido definida

El preprocesador de C ofrece directivas condicionales que permiten seleccionar condicionalmente código fuente.

#ifdef, #ifndef , #endif

Las directivas **`#ifdef y #ifndef`** inquieren por la definicion de una macro, su forma general es:

```
#ifdef nombre_de_macro
        bloque de sentencias
#endif
```

Si se ha definido la macro especificada entonces se ejecuta el bloque de sentencias englobado hasta el **`#endif.`**

O bien su contraparte :

```
#ifndef nombre_de_macro
        bloque de sentencias
#endif
```

Si la macro no ha sido definida entonces se ejecutan las sentencias comprendidas entre el **`#ifndef`** y **`#endif`**

En un sistema multiarchivo no se permite la declaración múltiple para un solo macro. Nuestro macro ESC para definir la tecla de salida, solo podría estar definida en uno de los archivos. Sin embargo en un sistema grande, o con fines de depuración, podría ser necesario definirlo en varios lugares a la vez. Para evitar los errores que acarrearía la declaración múltiple se puede realizar la siguiente definición en cada uno de los archivos que requieran la macro.

```
#ifndef ESC
      #define ESC 27
#endif
```

El primer archivo que el preprocesador encontrara realizaría la definición de la macro dado que esta no ha sido definida en ninguna otra parte. Los siguientes declaraciones del mismo tipo omitirían la directiva **`#define`** porque se detectaría que la macro ya ha sido declarada por vez primera en algun otro lugar del sistema.

El siguiente ejemplo despliega un mensaje solo si la macro **`DEPURACION`** ha sido definida.

```
#ifdef DEPURACION
      printf("\n El valor de la variable es: %d",variable);
#endif
```

Estas directivas se pueden anidar al menos hasta en un nivel de ocho directivas.

`#error`

Esta directiva fuerza al compilador a desplegar un mensaje de error en la pantalla para posteriormente detener la compilación. El mensaje no debe ir entre comillas. . La forma general es:

```
#error mensaje_de_error
```

Esta directiva es sumante útil en el proceso de depuración.

#if, #elif, #else y #endif

La forma general para usar una directiva condicional en el preprocesador de C es:

#if expresion-constante
bloque verdadero de sentencias
#else
bloque falso de sentencias
#endif

Si la expresión constante es cierta, se compila el bloque de código correspondiente a la rama veradadera, de lo contrario compilará el bloque de código correspondiente al **#else** asociado. Se utiliza la notación *expresión-constante* porque no es posible preguntar por valores variables. La expresión deberá estar formada por una expresión condicional que relacione dos constantes. O bién, por una macro que almacene un valor de verdadero o falso.

Ejemplo:

Si se desea procesar un bloque específico como encabezado dependiendo del tipo de compilador usado se puede realizar la siguiente pregunta

```
#if __STDC__
      #include <ansi.h>
#else
      #include <kernigal.h>
#endif
```

Ejemplo:

Para una aplicación determinada en una empresa con diversos tipo de equipos y distintos monitores se podría escribir el siguiente segmento de código.

```
/* Definición de las características del equipo */
#define VGA  1
#define CGA  0
.
.
.
#if VGA
      printf("\n Inicializando gráficos de video VGA ");
      ...
#else
      #if CGA
            printf("\n Preparando tarjeta de video CGA ");
            ...
      #else
            #error No existe controlador de video para esa configuracion
      #endif
#endif
```

La directiva #elif suministra la selección múltiple. Es equivalente a realizar un anidamiento de decisiones. Su forma general es:

```
#if expesion
        bloque de sentencias
#elif
        bloque de sentencias
#elif
        bloque de sentencias
.
.
.
#endif
```

El ejemplo precedente se podría reescribir de la siguiente manera.

```
/* Definición de las características del equipo */
#define VGA  1
#define CGA  0
.
.
.
#if VGA
      printf("\n Inicializando gráficos de video VGA ");
      ...
#elif CGA
      printf("\n Preparando tarjeta de video CGA ");
      ...
#else
      #error No existe controlador de video para esa configuracion
#endif
```

Ejemplo:

En base al país en donde operará el sistema se desplegará el símbolo monetario correspondiente:

```
/* Notación internacional */
#define MEX       1
#define USA       2
#define GB        3
.
.
#define PAIS      MEX
#if PAIS = = MEX
      char sim_monet[]="N$";
#elif PAIS = = USA
      char sim_monet[]="$";
#elif PAIS = = GB
      char sim_monet[]="£";
#else
      #error Las opciones para el pais no se encuentran definidas
#endif
```

En este ejemplo se puede apreciar que las expresiones que se utilizaron en las estructuras de desición involucraron solamente a valores constantes. Antes del proceso de compilación, el programador modificará el pais para el cual desee compilar su aplicación, pero una vez hecho esto, no podrá modificar el valor y por tanto es considerado técnicamente como una constante.

II.3.8 Programas multiarchivo.

La modularidad de los programas y un buen diseño de las funciones que conforman un sistema, nos permitirán crear librerias de rutinas que podamos utilizar tantas veces como sea necesario. La reutilización de código abate en gran medida los costos de desarrollo y permite al programador dedicarse a la creación de software siempre nuevo, en vez de reescribir código ya existente. Generalmente se trata solo de ligeras variaciones para aplicarla en el sistema que lo ocupa en un momento determinado.

La creación de dichas librerias se puede realizar colocando las funciones que conforman un conjunto de tareas relacionadas (video, impresión, despliege de cajas de dialogo y ventanas, manejo de archivos, manejo de funciones de red, etc) dentro de archivos separados y posteriormente compilandolos con lo que es la parte diferente y especifica del sistema en desarrollo.

Los programas escritos en lenguaje C suelen ser escritos de esta manera. El uso de multiples archivos implica que se deba especificar al compilador cuales son archivos que se utilizarán para obtener todo el código fuente necesario para un sistema.

La directiva **#include** ha sido utilizada a todo lo largo de estos apuntes. Era requerida para poder compilar el archivo de cabecera stdio.h para que los programas pudieran ser correctamente compilados.

La forma general de la directiva #include es:

#include <archivo>

O su forma alterna:

#include "ruta_de_acceso_archivo"

En los compiladores de C suelen definirse ciertos directorios de trabajo en donde el compilador buscará los archivos que requiera para la compilación. El directorio INCLUDE especifica el lugar en donde el compilador buscara los archivos definidos por la directiva #include con corchetes angulares. Cuando se utilizan dichos corchetes se está indicando que los archivos se encuentran dentro del directorio especificado para los archivos include. En el sistema UNIX por omisión dicho directorio es **/usr/include**. En los sistemas orientados a PC el sistema integrado de programación inquiere por dicho directorio a través de una ventana de diálogo.

Cuando se usa la directiva #include con comillas se indica al compilador que el archivo especificado se encuentra en el directorio de trabajo o en la ruta de acceso descrita.

Ejemplo:

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

Busca el archivo **stdio.h** en el directorio **/usr/include** para equipo UNIX, o en el directorio \tc\include para equipos PC

```
#include "ventanas.h"
```

Busca el archivo en el directorio de trabajo, de no encontrarlo, busca en el directorio INCLUDE.

```
#include "d:\lib\c\red.c"
```

Busca el archivo red.c en la ruta especificada.

Existe una forma alternativa establecida por el estandar ANSI para definir un archivo include, esto es utilizando macros. Vease el siguiente ejemplo

```
/* Colocar a uno si se compila en UNIX
   Colocar a cero si se commpila en PC*/
#define UNIX 1
#if UNIX
      #define STD_INCLUD       "/USR/INCLUDE/stdio.h"
#else
      #define STD_INCLUD "\TC\INCLUDE\stdio.h"
#endif

#include STD_INCLUD
```

Preguntas:

1.- ¿Cuales son las partes constitutivas de una función en lenguaje C?
2.- Con sus propias palabras defina lo que significa lista de parámetros
3.- ¿Qué ventajas representa el hecho de que el lenguaje C acepte y retorne valores desde y hacia el sistema operativo?
4.- ¿Cuales son las dos formas de declarar los parámetros de una función en C?
5.- ¿Porqué es importante especificar el tipo de resultado que entrega una función?
6.- ¿Qué es el preprocesador?
7.- ¿Qué es una directiva?
8.- ¿Qué es una macro?
9.- ¿Cuales son las ventajas que encuentra en la existencia del preprocesador?
10.- Observe las directivas de preprocesador. ¿Cumplen con el teorema de la estructura?¿Qué concluye con ello?
11.- ¿Qué relación tienen los conceptos de acoplamiento con la posibilidad de realizar compilación multiarchivos?
12.- ¿Podría indicar como implantar cada uno de los distintos tipos de acoplamiento vistos en el capitulo uno?
13.- ¿De que forma, el preprocesador podría hacer menos legible un programa?
14.- ¿Cuales son los riesgos potenciales de abusar de las macros?

II.4 Estructuras, uniones y punteros.

Las estructuras de control de un programa se ven complementadas con las estructuras de datos que un lenguaje de programación proporciona y las que el mismo programador puede llegar a definir. El diseño de las estructuras de datos queda fuera de los alcances de estos apuntes.

Lenguaje C permite manipular los tipos simples de datos para crear estructuras más complejas basadas en dichos tipos primigenios.

II.4.1 Estructuras y uniones

ESTRUCTURAS

El lenguaje C, así como la mayoría de los lenguajes de programación, proporciona un conjunto finito de tipos de datos en base a los cuales se pueden realizar programas de regular complejidad. El modelado de los problemas de la *vida real* nos lleva a requerir de tipos de datos distintos. En cualquier aplicación nos encontraremos que un dato particular estará compuesto por campos individuales que por sí solos no poseen significado.

Pongamos el caso de la información necesaria para almacenar los datos de una persona: nombre, domicilio, fecha de nacimiento. Cada uno de estos datos puede descomponerse en campos que conforman al dato. Desglozando cada uno de los ejemplos propuesto tendríamos:

nombre: (nombre_pila , apellido_paterno, apellido_materno)

domicilio: (calle, numero, colonia, población, estado, codigo_postal)

fecha-nacimiento: (dia, mes, año)

A su vez estos datos pueden unirse para formar un ítem superior:

empleado: (nombre, domicilio, fecha_nacimiento)

El lenguaje C permite formar estas estructuras a través del especificador `struct` tiene la siguiente forma general:

```
struct nombre_estructura
{
        tipo identificador ;
        tipo identificador ;
        tipo identificador ;
        .
        .
        .
        tipo identificador ;
} lista de variables_de_estructura;
```

Donde:

nombre_estructura: Representa el nombre que recibira la estructura. Dicho nombre será único para la estructura.

identificador: Es el nombre de un componente de la estructura, cada componente puede ser de tipo elemental o compuesto.

variable_de_estructura: Es el nombre de una variable que estará conformada por los tipos declarados en la estructura.

Se puede omitir el *nombre_estructura* o bien la *lista de variables de estructura* pero no ambas. Observese que la definición lleva punto y coma; la declaración es una sentencia. Cada componente de la estructura recibe el nombre de *miembro de la estructura.*

Utilizando la definición enunciada hace un momento la estructura **nombre** se puede declarar de la siguiente manera:

```
struct nombre
{
      char nom_pila[20];
      char a_paterno[20];
      char a_materno[20];
}identidad;
```

Se ha declarado una estructura que llevará el nombre genérico de **nombre** y que estará constituido por tres miembros de tipo caractér. La declaración incluye una variable del tipo definido por la estructura que recibirá el nombre de **identidad.**

Una forma altema de realizar una declaración de estructura es la siguiente

```
struct fecha
{
int dia;
char mes[3];
int año;
};

struct fecha f_nacimiento, f_ingreso, f_baja;
```

La declaración se ha realizado en dos sentencias separadas. En la primera se define una estructura con los tres miembros que conforman a una fecha. Hasta ese momento no se ha declarado *ninguna* variable que corresponda a esa estructura. Dicha declaración se realiza hasta la segunda sentencia en donde se utiliza el nombre de la estructura para declarar a la variable. Dado que todas las fechas del sistema se regirán con dicho formato, basta con declarar la estructura en forma global y despúes declarar cada variable en forma local.

Esto no contradice los principios expuesto hasta este momento. La definición de la estructura no declara variables, pero es visible en todas partes en donde sea requerido declarar variables locales. Las variables serán todas de tipo local. Si en un momento dado se decide que la estructura del mes pasa de ser de tres a once caracteres, la actualización será general en toda la aplicación con la modificación de una sola linea. Teniendo cuidado de los efectos secundarios que esto pueda acarrear.

Una tercera forma de declara una estructura es no asignarle nombre. Esto puede suceder cuando no se requiera mas que una variable con la estructura declarada. Por ejemplo:

```
struct
{
      char calle[10];
      int numero;
      char colonia[20];
      char poblacion[20];
      char estadop[15];
      unsigned int cod_postal;
}domicilio;
```

La estructura no recibe nombre, pero ha sido declarada la variable **domicilio** que consta de los seis miembros enunciados.

La declaración de estructuras con miembros más complejos se realiza exactamente de la misma manera, por ejemplo, utilizando las estructuras definidas hasta este momento podemos declarar:

```
struct empleado
{
      struct nombre identidad;
      struct fecha f_nacimiento;
      struct fecha f_ingreso;
      struct fecha g_baja;
      char cve_depto[10];
}obrero.
```

Para accesar a cada uno de los miembros de la estructura es necesario utilizar la siguiente forma general:

variable_de_estructura . miembro_de_estructura

Por ejemplo para asignar un valor desde el teclado al miembro **nom_pila** del la variable **identidad** se realizara de la siguiente forma:

```
gets(identidad.nom_pila);
```

Tal como se puede apreciar solo se agrega el nombre de la estructura definida y se separa este nombre con el nombre del miembro a través del *operador de selección de miembro* **(.)**. Así mismo una asignación a un elemento de estructura recibe el mismo tratamiento.

Ejemplo:

```
struct  vector
{
      float modulo;
      float angulo;
}vect1;

vect1.modulo = 15.102;
vect1.angulo = 30.6;
```

Una estructura es suceptible de ser subindizada para formar arreglos de estructuras. Como en cualquier arreglo, el primer subíndice es cero. La forma para realizar una declaración puede ser cualquiera de las variantes expuesta previamente, aunque es más común utilizar la siguiente:

struct nombre_estrcutura variable[longitud];

Por ejemplo para declarar un arreglo 100 elementos de la estructura **nombre** que hemos definido con anterioridad se escribiría una sentencia como la siguiente:

```
struct nombre identidad[100];
```

El acceso a cada uno de los elementos de la estructura se realiza subindizando el nombre de la estructura:

```
gets(identidad[3].nom_pila);
```

Es muy importante notar la gran diferencia que existe entre subindizar el nombre de la estructura y subindizar el miembro de la estructura. En el ejemplo propuesto se ha declarado un arreglo de 100 elementos de estructura que reciben el nombre de identidad. A su vez el miembro **nom_pila** consta de veinte caracteres.

```
identidad[3].nom_pila='\0';        /* Coloca en nulo el nombre de pila del
                                      tercer elemento del arreglo de
                                      identidad */

identidad.nom_pila[3]='\0'        /* Coloca un nulo en el tercer caracter del
                                      miembro nom_pila */
```

En versiones antiguas de C no era permitido realizar asignaciones de estructuras. Por ejemplo:

```
struct coord
{
      int x;
      int y;
};

struct coord sup_izq, inf_der;

sup_izq.x=0;
sup_izq.y=0;

inf_der = sup_izq;
```

En este ejemplo se declara una estructura para definir coordenadas en la pantalla. Las variables **sup_izq** y **inf_der** son del tipo **coord**. En dos sentencias se asignan valores a cada uno de los miembros de la estructura para concluir este segmento con la asignación de la estructura **sup_izq** en la estructura **inf_der**. Aunque no se ha accesado a los elementos de **inf_der** en forma individual, tanto **inf_der.x** como **inf_der.y** están ahora inicializados en cero.

Recuerde que esta es una característica de C ANSI, los compiladores que no se ajusten a este estándar no aceptarán dicha sentencia.

Problema:

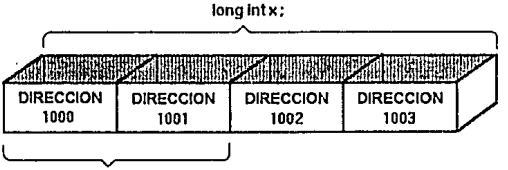
Un profesor de ENEP ARAGON requiere de un sistema que contenga el nombre del alumno y las tres calificaciones con las que es evaluado a lo largo del semestre. Los grupos suelen ser de 50 alumnos. Escriba un programa que obtenga los promedios de los alumnos

UNIONES

En el epígrafe I.3.1 se mencionó que existen diversos tipo de acoplamientos de datos. Uno de ellos es el acoplamiento por zonas de datos compartidas. La union es un buen ejemplo de este tipo de acoplamiento.

En una unión se hace uso de las mismas localidades de memoria para un conjunto determinado de variables con nombres distintos. Generalmente este tipo de uniones reunen a datos de tipo distinto

En la memoria de una computadora, los datos son almacenados en localidades de memoria asignados en el tiempo de ejecución. Un dato de tipo entero ocupa por lo común menos espacio que un número de tipo real. Dentro de una unión , dos o más variables pueden compartir la misma localidad de memoria bajo diferentes nombres e inclusive bajo distintos tipos de datos.



Unión de una variable entera con una variable entera larga

La forma general para realizar una union es la siguiente

```
union nombre_union
{
        tipo identificador;
        tipo identificador;
        tipo identificador;
        .
        .
        .
        tipo identificador;
} lista_de_variables;
```

El ejemplo mostrado en el gráfico se declararía como :

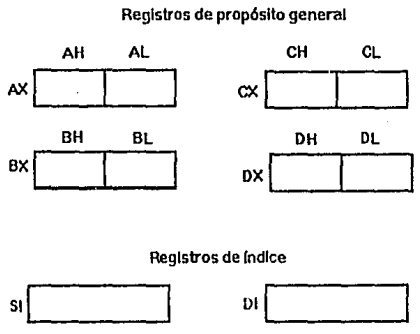
```
union ejemplo
{
        long int x;
        int y;
}
```

INTERRUPCIONES A LA FAMILIA DEL MICROPROCESADOR 8086

Bajo los entornos basados en DOS, las interrupciones al microprocesador suelen ser muy comunes en los programas profesionales.

La familia del microprocesador 8086 soporta instrucciones de interrupcion de programas. Una interrupción detiene momentaneamente la ejecución de un programa para ejecutar diversas rutinas de sistema. Dichas rutinas acceden directamente a la localidades de memoria para el uso exclusivo del mircroprocesador. Dicha localidades de memoria se encuentran en el mismo microprocesador y reciben el nombre de *registros*.

Los registros del 8086 tienen diversos nombres. Los que se utilizan en la interrupciones son mostrados a continuación.



Como se puede observar, el registro **AH** está conformado por los segmentos **AH** y **AL**. Lo mismo es verdadero para **BX** (formado por **BH** y **BL**) , **CX** (formado por **CH** y **CL**) y **DX** (formado por **DH** y **DL**).

Los compiladores de DOS incluyen el archivo de cabecera `dos.h` en el cual se encuentra la definición de dos estructuras que definen a los regisros desde los dos puntos de vista mostrados en el gráfico. La primera toma los registros de microprocesador como palabras de dos bytes. La segunda estructura lo define como palabras de un solo byte. Finalmente la declaración de la unión asegura que ambas estructuras hagan referencia a los mismos registros.

El segmento de cabecera de **dos.h** tal y como lo define Borland es listado a continuación.

```
/*    dos.h

      Define estructuras, uniones, macros, y funciones para utilizar
      con MSDOS y la familia del microprocesador Intel iAPX86.

        Copyright (c) Borland International 1987,1988,1990
      Todos los derechos reservados.
*/
struct WORDREGS {
      unsigned int      ax, bx, cx, dx, si, di, cflag, flags;
};

struct BYTEREGS {
      unsigned char     al, ah, bl, bh, cl, ch, dl, dh;
};

union REGS  {
      struct      WORDREGS x;
      struct      BYTEREGS h;
};
```

Una interrupción al microprocesador utiliza la función **int86()**. Dicha interrupción tiene la forma general:

```
#include <dos.h>
int int86(int numero_interrupcion, union REGS *entrada, REGS *salida);
```

Donde:

numero_interrupcion : Es el número de la rutina a la que llamará
*entrada: Es un puntero a una union del tipo REGS que contendrá los datos de entrada para realizar la interrupcion.
*salida: Es un puntero a una union del tipo REGS que contendrá los resultados de la interrpcion

Algunas de la interrupciones más importantes son listadas a continuación

Interrupcion	Función
5h	Impresión de pantalla
10h	Entrada/ Salida de video
11h	Lista de equipo
12h	Tamaño de memoria
13h	Entrada/Salida de disco
14h	Entrada/Salida del puerto serie
16h	Entrada/Salida del teclado
17h	Entrada/Salida de la impresora
1Ah	Hora y fecha

Los números de interrupción se encuentran definidos en numeración hexadecimal.

Cada una de la interrupciones utilizan los registros de diversas formas. La lista que se da a continuación no trata exahustivamente todas las opciones, para ello es necesario referirse al manual del compilador que se esté utilizando.

Interrupcion 10H . Funciones de video

Contenido del registro AH	Funcion realizada
0	Fijar el modo de video. AL = modo de video
1	Fijar el numero de lineas del cursor CH: bits 0-4 con la linea inicial bits 5-7 a cero. CL: bits 5-4 con la linea final bis 5-7 a cero.
2	Coloca el cursor en una posicion determinada DH: Renglon DL: Columna BH: Página de video
3	Obtiene la posición del cursor BH: Número de página de video Devuelve: DH: Renglón DL: Columna CX: Modo de video
4	Obtiene la posicon del lápiz óptico Devuelve: AH=0 con el lápiz inactivo AH=1 con el lápiz activo DH: Renglón DL: Columna CH: Linea actual (0-199) BX: Columna del pixel (0-139 ó 0-639)
5	Especifica la página de video activa AL: debe encontrarse entre 0 y 7
6	Desplaza la página hacia arriba AL: número de lineas a desplazar. Con cero desplaza todas CH: Renglón de la esquina superior izquierda CL: Columna de la esquina superior izquierda DH: Renglón de la esquina inferior derecha DL: Columna de la esquina inferior derecha BH: Atributo para ser usado por la líneas en blanco.
7	Desplazar la página hacia abajo Los registros se utilizan de la misma manera que en el punto anterior
8	Obtiene el caracter en la posicion del cursor BH: Página de video Devuelve AL: Caractér leido AH: Atributo.

Interrupcion 10H . Funciones de video (continuación)

Contenido del registro AH	Funcion realizada
9	Escribir caracter y atributo en la posición del cursor BH: página de video BL: Atributo CX: número de caracteres a ecribir AL: caractér
Ah	Escribir caractér en la posición actual del cursor BH: Página de video CX: Número de caracteres a escribir AL: caractér
Bh	Especificar la paleta de colores BH: numero de la paleta BL: color
Ch	Escribir un punto DX: número de renglón CX: número de columna AL: color

Interrupcion 10H . Funciones de video (continuación)

Contenido del registro AH	Funcion realizada
DX:	Obtiene un punto DX: Número de renglón CX: Número de columna Devuelve: AL: Punto teído
Eh	Escribir un caractér en la pantalla y avanzar el cursor AL: Caractér BL: color de primer plano BH: página de video
Fh	Obtiene el estado del video Devuelve AL: modo de video AH: Número de columnas de pantalla BH: Página de video activa actualmente.

TABLA DE MODOS DE VIDEO

Modo	Tipo	Dimensiones	
		Gráficas	Texto
0	Texto,B/N	no disponible	40 x 25
1	Texto 16 colores	no disponible	40 X 25
2	Texto B/N	no disponible	80 X 25
3	Texto 16 colores	no disponible	80 x 25
4	Gráficos 4 colores	320 x 200	40 x 25

TABLA DE MODOS DE VIDEO (continuación)

Modo	Tipo	Dimensiones	
		Gráficas	Texto
5	Gráficos 4 tonos de gris	320 x 200	40 x 25
6	Gráficos 2 colores	640 x 200	80 x 25
7	Texto B/N	no disponible	80 x 25
10	Gráficos 4 colores	320 x 200	40 x 25
13	Gráficos 16 colores	640 x 200	80 x 25
14	Gráficos 2 colores	640 x 350	80 x 25
15	Gráficos 16 colores	640 x 480	80 x 25
16	Gráficos 2 colores	640 x 480	80 x 30
17	Gráficos 16 colores	640 x 480	80 x 30
18	Gráficos 256 colores	640 x 200	40 x 25

Programa:

Hacer una función para colocar el cursor en una posición de la pantalla. Utilice la interrupción de video 10h.

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

#define VIDEO 0x10


int main(void)
  {
     clrscr();
     cursor_xy(35, 10);
     printf("¡ Hola !");
}

void cursor_xy(int x, int y)
  {
     union REGS regs;

     regs.h.ah = 2;  /* Coloca la posicion del cursor */
     regs.h.dh = y;
     regs.h.dl = x;
     regs.h.bh = 0;  /* Página de video */
     int86(VIDEO, &regs, &regs);
  }
```

Interrupcion 11H . Lista de equipo

Bits del registro AH	Funcion realizada
	Obtiene lista del equipo conectado al CPU Devuelve
0	Presencia de discos flexibles
1	No usado
2,3	RAM de la tarjeta de sistema 11= 64 K
4,5	Modo de video inicial 01: 40 columnas 10: 80 columnas color 11: monócromo
6,7	Número de unidades de disco
8	Chip de DMA instalado 0 = instalado
9,10,11	número de puertos RS-232
12	Adaptador de juegos instalado 1=instalado
14,15	Número de impresoras

Interrupcion 13H . Funciones de Entrada/Salida de disco

Contenido del registro AH	Funcion realizada
0	Reinicializar el sistema de disco
2	Leer sectores en memoria DL: Número de memoria DH: Número de cabeza CH: Número de pista CL: Número de sector AL: Número de sectores a leer. ES:BX: Dirección del buffer Devuelve: AL: Número de sectores leídos AH: 0 si no hubo errores
3	Escribir sectores en el disco: Los registros se definen igual que en el punto dos
4	Verificar Los registros se definen igual que en el punto dos
5	Formatear una pista DL: Número de unidad DH: Número de cabeza CH: Número de pista ES:BX: Información del sector

Interrupcion 12H . Tamaño de la memoria RAM

Registro AH	Funcion realizada
	Devuelve AX: número de kilobytes de RAM

Interrupcion 16H . Funciones del teclado

Contenido del registro AH	Funcion realizada
0	Obtiene código de inspección Devuelve: AH: Código de inspección AL: Código del caractér
1	Obtiene el estado del buffer: Devuelve ZF: 1 si el buffer está vacio 0 cuando hay caracteres esperando con el siguiente caractér en AX como antes

Interrupcion 17H . Funciones de impresora

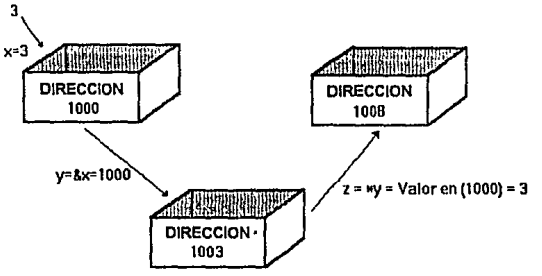
Contenido del registro AH	Funcion realizada
0	Imprimir un caractér AL: caractér DX: número de impresora Devuelve AH: Estado
1	Inicializar impresora DX: Número de impresora Devuelve AH: Estado
2	Obtiene estado de la impresora DX: Número de impresora Devuelve AH: Estado

Volveremos a tratar con las interrupciones en el apartado II.6 en donde se explicará con más detalle la programación a nivel de bits.

II.4.2 Apuntadores a datos simples

Una variable que almacena una dirección recibe el nombre de **puntero** ya que es un apuntador a una localidad de memoria que es distinta a ella misma. Un puntero se declara indicando el tipo de dato al que apuntará y se le antepone un asterisco. Para obtener la dirección que ocupa una variable se utiliza el operador ampersand (&). Dicho operador también es unario y prefijo.

Observemos el siguiente gráfico.



Uso de los operadores de dirección

Sea una variable x localizada en la dirección 1000. La variable x puede recibir en una asignación del tipo `x=valor` un número cualquiera. El valor contenido en la dirección 1000, para el ejemplo del gráfico es igual a tres. Una variable conocida por el programador como y se encuentra localizada en la localidad 1003. Dicha variable y recibe la *dirección en donde está localizada la variable* x. El contenido de la dirección 1003 es la dirección de la variable x. Es decir, la variable y apunta a la dirección x.

Si lo que deseamos obtener es *el valor contenido en la dirección a la cual apunta la variable y*, hacemos uso del operador asterisco. La variable z la suponemos localizada en la dirección 1008. La expresión `z=*y` da por resultado que el programa obtenga la dirección contenida en y, posteriormente obtiene el valor que se encuentra almacenado en dicha dirección. El resultado final será el número tres asignado a la variable z.

En el lenguaje C una variable puntero debe especificar el tipo de dato al cual apuntará. La forma general para declarar una variable puntero es la siguiente:

*tipo *identificador, *identificador, *identificador;*

Veamos el siguiente ejemplo:

`int numero;`	Esta instrucción asigna una localidad a la variable **numero** Supongamos que la dirección asignada es 3AFF
`int *puntero`	Esta instrucción crea un **puntero** a un entero.
`numero=10;`	Esta instrucción asigna el valor de 10 a la dirección 3AFF.
`puntero=&numero`	Aquí se asigna 3AFF como valor a **puntero** .
`numero=*puntero+2;`	Al valor almacenado en la dirección apuntada por **puntero** sumale dos unidades.
`printf("%d",numero);`	Imprime el valor de **numero** que es 12.

La declaración para los tipos de datos básicos se realiza como hemos visto anteriormente. Un tipo de dato más complejo como lo es una arreglo recibe una trato ligeramente distinto. Cuando se realiza una declaración de un arreglo, el nombre del arreglo en sí mismo es un puntero a la primera localidad de memoria que ocupa un arreglo.

Un puntero puede asignar directamente a otro puntero. Por ejemplo se puede realizar la siguiente serie de instrucciones en un programa:

```
float miexamen;
float *uno, *otro;

miexamen = 3.1;
uno = &miexamen;
otro = uno;
miexamen = miexamen + *uno + *otro;

printf (" \t Resultado de mi examen: %3.1f", miexamen);
```

En este caso la varible miexamen recibe el valor de 3.1. Las siguientes dos instrucciones obtienen la dirección de la variable miexamen. El resultado desplegado por la instrucción `printf()` será por tanto:

```
Resultado de mi examen: 9.3
```

II.4.3 Apuntadcres a datos compuestos

Se recordará que en un arreglo, cada elemento se diferencia del resto de los elementos por un subíndice. Dicho subíndice indica cuantos elementos existen entre él y el primero elemento del arreglo. Para generar un puntero a un arreglo solo de debe declarar una variable del mismo tipo que el arreglo al cual se apuntará.

```
int *puntero;
int arreglo[30];

puntero = arreglo;
/* una forma alterna de hacerlo es
puntero = &arreglo[0];   */
```

Se observa que en forma alterna se puede realizar la asignación ocupando la dirección del primer elemento del arreglo. Esto no es muy común en un programa escrito profesionalmente.

Las siguientes asignaciones son idénticas:

```
int x;
int arreglo[3] = {1, 4, 6};
.
.
.
x = arreglo[2] ;
x = *(arreglo+2);
```

La primera asignación toma el valor en `arreglo[2]` y lo asigna a la variable `x`. La segunda asignación obtiene el valor contenido en la dirección que se encuentra dos elementos después del primer elemento del arreglo. La primera forma de realizar la asignación es una forma abreviada de escribir lo que ejecuta la segunda sentencia.

Es muy importante notar que cuando se requiere trabajar con el apuntador a un arreglo, basta con ocupar el nombre del arreglo, Cuando se le incluyen los corchetes ya no se hace referencia a un puntero sino al valor de una variable. En la función `scanf()` es preciso anteponer un ampersand a toda variable simple. En el epígrafe II.4.4 se explicará con más detalle las razones por las cuales ocurre esto, Con ello se envía la dirección en donde `scanf()` colocará el valor obtenido del dispositivo de entrada para poder realizar una lectura correcta . Esto es cierto inclusive para leer un elemento de un arreglo. Pero cuando se desea leer una cadena (un tipo especial de arreglo), no se utiliza el ampersand porque ya se está utilizando el valor de dirección.

ASIGNACIÓN DINÁMICA DE MEMORIA

En la mayoría de los compiladores los programas generados solo pueden utilizar la memoria que el programa requiere. Esta memoria es asignada en tiempo de compilación. Es decir, antes de que el programa se empiece a ejecutar. En los ejemplos escritos hasta el momento, los arreglos tienen una dimensión finita que el programa no puede rebasar , aunque la aplicación exija mayor memoria y esta se encuentre libre en la máquina. Si en un instante cualquiera un programa en C tuviera requerimientos de memoria mayores a los previstos por el programador, o en una aplicación en donde la cantidad de memoria es indefinida (como en un procesador de texto), el programa puede realizar una petición de memoria para poder hacer uso de ella. Dicha asignación se realiza en tiempo de ejecución.

En este aparatado usaremos dos funciones que ayudarán a crear programas más flexibles. Son una característica muy poderosa del lenguaje C la cual permite realizar una petición de memória.

El operador monario `sizeof` obtiene el número de bytes ocupado por un tipo de dato o por una variable determinada.

Por ejemplo si deseamos saber cuantos bytes ocupa una variable flotante podríamos tener el siguiente segmento de código.

```
float var;

printf ("%d", sizeof(var));
```

Con lo cual se desplegará el valor deseado.

Dicho valor es importante cuando deseamos realizar una petición de memoria de un arreglo. Generalmente sabremos cuantos elementos deseamos obtener, pero no la cantidad de memoria que estos elementos ocupará .

La función `malloc()` (memory allocation = localización de memoria) realiza una petición de memoria . La función prototipo de la malloc() es la siguiente:

#include < stdlib.h>
*void *malloc(size_t número_de_bytes);*

Donde:

size_t	Se encuentra definida en *stdlib.h* como un entero sin signo
numero_de_bytes	Es la cantidad en bytes que deseamos obtener de memoria

Cuando una petición de memoria no resulta con éxito la función `malloc()` retoma un valor nulo. Es absolutamente necesario que después de cada petición de memoria se verifique que la petición ha resultado exitosa, de esa manera se evitarán problemas.

La contraparte de la función `malloc()` es la funcion `free()` (free= libre) la cual libera la memoria. Su forma general es

#include < stdlib.h>
*void free (void *puntero)*

Donde:

puntero	Apunta a la primer localidad de memoria que se desea liberar.

Ejemplo:

Realizar un programa que lea una cadena. La asignación de la cadena debe ser en forma dinámica

```
/* ejemplo II.4.3.1

     Asignación dinámica de la memoria   */

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

#define LONGITUD 30

void main(void)
{
char *cadena;
int contador;

/* Realiza la asignación */
cadena = malloc(sizeof( char )*LONGITUD );
clrscr();
```

```
/* Verifica que no sea nulo */
if ( cadena )
{
      printf("\n Digite una cadena: ");
      gets(cadena);
      printf(" \n La cadena fué: %s", cadena);
      free (cadena);

}
else
{
      printf("\n Memoria insuficiente");
      printf("\n Libere memoria cerrando algunas aplicaciones");
}
}
```

Resultados:

```
Digite una cadena: E.N.E.P

La cadena fué: E.N.E.P
```

Observaciones:

Se hace uso de la macro `LONGITUD` para definir el número de elementos del arreglo que será asignado dinámicamente. Cada elemento ocupa una cantidad determinada de bytes de memoria, de tal forma que para realizar la petición de memoria correcta es necesario multiplicar el número de elementos del arreglo por la catidad de bytes que cada elemento requiere.

La función `maloc()` devolverá una dirección de memoria válida si la petición resulta exitosa. El if que se encuentra a continuación de la petición verifica que esto ocurra para poder realizar el bloque que utiliza la memoria asignada. La función `free()` liberará estas localidades cuando el proceso termine.

Debe tenerse cuidado con la función `free()` ya que si por error toma una dirección inválida, se provocarán conflictos en la memoria RAM y la máquina terminará por detenerse o dañar información.

ESTRUCTURAS LIGADAS

Aunque el tema de las estructuras está más alla de los límites de estos apuntes, mostraremos una forma sencilla de realizar listas ligadas utilizando estructuras ligadas asignadas dinámicamente.

Tal y como se había mencionado, un puntero debe ser declarado para apuntar a un tipo de dato específico. Dicho dato, lo mismo puede ser un entero que toda una estructura. La forma general sigue siendo la misma. Por ejemplo, para crear un puntero a la estructura `nombre` definida en le epígrafe II.4.1 solo se tiene que escribir el tipo de dato y el identificador del puntero. Observemos el siguiente ejemplo:

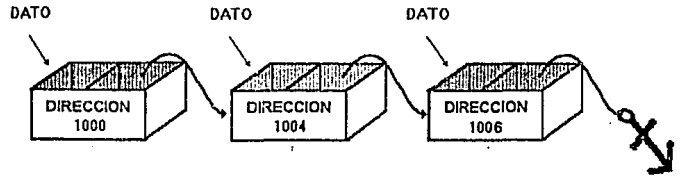
```
struct nombre
{
      char nom_pila[20];
      char a_paterno[20];
      char a_materno[20];
}identidad;
struct nombre *puntero;

puntero = &identidad;
```

La estructura **nombre** está conformado por los miembros **nom_pila, a_paterno** y **a_materno**. La variable **identidad** es del tipo de la estructura **nombre**. Posteriormente se declara que la variable **puntero** es un apuntador a una estructura del tipo **nombre**. Para realizar la asignación de la dirección de la variable **identidad** a **puntero**, simplemente nos valemos del operador ampersand. Para accesar a los miembros de la estructura se hace uso del operador flecha -> (un signo menos seguido de un signo mayor). Para accesar al elemento conteniendo el nombre de pila se utilizaría una asignación del siguiente tipo:

```
persona = puntero->nom_pila;
```

Cuando se accesa a una estructura directamente se utiliza el operador punto. Cuando se accesa por medio de un puntero se hace uso del operador flecha. Esta distinción se debe tener muy presente cuando se escriban programas.

Un miembro de una estructura puede ser una apuntador a otra estructura. Cada estructura posee información relativa a un dato particular. A través de la asignación dinámica de memoria podemos ir creando una lista de estructuras en las cuales cada una apunte a la que le sigue. Observese el gráfico.



En el ejemplo se tiene una estructura cualesquiera, su localidad reside en la dirección 1000 de la memoria. Esta estructura tiene un miembro que es un puntero a otra estructura, de tal forma que la estructura de la localidad 1000 apunta a otra estructura localizada en la dirección 1004. Esta a su vez apunta a una tercer estructura alojada en la dirección 1006. Por último, dicha celda apunta a *ningún lugar* (conocido como *tierra*) y la lista termina.

Cada estructura recibe el nombre de *nodo de información*. Al conjunto de nodos organizados a través de una serie de apuntadores, en los cuales un nodo apunta a otro nodo en forma lineal se le conoce como *lista*. El primer nodo recibe el nombre de *cabeza de lista*. Si un nodo no apunta a una localidad válida de la memoria se dice que dicho puntero apunta a *tierra*.

Si cada nuevo elemento es asignado dinámicamente, tendremos una lista que podrá crecer hasta donde la memoria libre lo permita.

Programa:
Realizar un programa que implemente un sencillo procesador de textos. El procesador deberá tener los siguientes comandos:

a) Un comando para insertar nuevas lineas al final del texto.
b) Un comando para borrar la última linea capturada
c) Un comando para listar todo el texto capturado

Pseudocódigo:

```
Captura de textos
LIMPIA PANTALLA
CREA cabeza de lista
ESCRIBE(" ?: Ayuda");
HAZ MIENTRAS (No se desee abandonar el programa)
      ESCRIBE EL PROMPT(">")
      LEE ( comando )
      HAZ CASO DE (comando)
       CASO DE comando sea insersion:
                  LLAMA RECORRE()
                  HAZ MIENTRAS( Se desee insertar )
                        REALIZA LA ASIGNACION DE MEMORIA AL nodo
                        SI (La asignacion fue exitosa) ENTONCES
                              INSERTA nodo->siguiente
                              LLAMA LECTURA()
                              ASIGNA tierra AL ULTIMO nodo
                              SI (SE DESEA DEJAR DE INSERTAR)
                                    BORRA nodo INSERTADO
                              FIN DEL SI
                        SINO
                              ESCRIBE(" Memoria insuficiente");
                        FIN DEL SI
      CASO DE comando sea baja:
                  LLAMA RECORRE()
                  SI (HAY nodo ES LA LISTA Y NO ES LA cabeza de lista)ENTONCES
                        ESCRIBE(texto )
                        PREGUNTA("Desea borrar? [S/N]")
                        SI(SE DESEA BORRAR )ENTONCES
                              LLAMA BORRAR()
                        FIN DEL SI
                  SINO
                        ESCRIBE(" Texto vacio")
                  FIN DEL SI

      CASO DE comando sea Listar:
                  ASIGNA primero A nodo
                  HAZ MIENTRAS ( EXISTA nodo )
                        ESCRIBE(texto)
                        MUEVE nodo AL nodo_siguiente
                  FIN DEL HAZ

      CASO DE comando sea ayuda:
                  DESPLIEGA AYUDA
      CASO DE comando sea salida:
                  LIBERA LA MEMORIA UTILIZADA

      FIN DEL CASO
FIN DEL HAZ
```

Solución:

```
/* ejemplo II.4.3.2

      Asignación dinámica de la memoria
      Estructuras ligadas                */

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

/* Definicion de las macros */
#ifndef ESC
      #define ESC 27
#endif
#ifndef ENTER
      #define ENTER 13
#endif
#ifndef REGRESO
      #define REGRESO '\b'
#endif
#define MAXLONG  80

/* Funciones prototipo */
void borrar(void);
void recorre(void);
void lectura(void);


/* Variables del sistema  */
struct linea
 {
      char texto[ MAXLONG ];
      struct linea *siguiente;
 } renglon;

int comando;
struct linea *proximo, *nodo, *anterior, *primero;

/* Programa principal */
void main(void)
{

 /* Limpia pantalla */
 clrscr();

/* Crea cabeza de lista */
 comando ='\n';
 renglon.texto[0] ='\0';
 renglon.siguiente = NULL;
 primero = &renglon;
```

```
printf("\n ?: Ayuda");
/* Ciclo principal */
while (comando != ESC)
{
      /* Prompt de comandos */
      printf("\n>");
      comando=toupper(getch());

      /* Selecciona el comando */
      switch (comando)
      {
            /* Insercion */
            case 'I':
                  printf("%c",comando);
                  recorre();
                  /* Mientras se desee insertar */
                  while ( comando != ESC )
                  {
                        /* Realiza la asignacion del nodo */
                        proximo = malloc(sizeof( renglon ) );


                        if (proximo)
                        {
                              /* Inserta nuevo nodo */
                              nodo->siguiente=proximo;
                              anterior = nodo;
                              nodo = proximo;

                              /* Lee texto */
                              printf("\n\t:");
                              lectura();

                              nodo->siguiente = NULL;
                              if (nodo->texto[0] == ESC)
                              {
                                    /*Si se desea salir
                                    borra nodo recien insertado */
                                    borrar();
                                    comando = ESC;
                              }
                        }
                        else
                        {
                              printf("\n\a Memoria insuficiente");
                              comando = ESC;
                        }
                  }
                  comando = NULL;
                  break;
```

```
            /* Borrar */
            case 'B':
                  printf("%c",comando);
                  recorre();

                  /* Si es nodo con informacion */
                  if (nodo && nodo != primero)
                  {
                        printf("\n\t %s",nodo->texto);
                        printf("\n\n\t\t Desea borrar? [S/N]");
                        comando = toupper(getche());
                        if ( comando == 'S'  )
                        {
                              borrar();
                        }
                  }
                  else
                  {
                        printf("\n\t\a Texto vacio");
                  }
                  break;
            /* Listar */
            case 'L':
                  printf("%c",comando);
                  nodo = primero;
                  /* Mientras exista un nodo */
                  while ( nodo)
                  {
                        printf("\n\t%s",nodo->texto);
                        nodo = nodo->siguiente;
                  }
                  break;

            /* Ayuda  */
            case '?':
                  printf("\n I Inserta texto");
                  printf("\n B Borra ultima linea de texto");
                  printf("\n L Lista todo el texto");
                  printf("\n ESC Salida");
                  break;

            /* Salida  */
            case ESC:
                        nodo = primero;
                  while ( nodo)
                  {
                              nodo = nodo->siguiente;
                        /* Libera todos los nodos insertados */
                        free(nodo);

                  }
      }
 }
}
```

Observaciones al programa principal:

Se define una macro que recibe el nombre de `MAXLONG` con la cual se especifica la longitud que tendrá cada uno de los renglones del texto que se capture. Basta modificar este parámetro para que el programa se actualice a un valor distinto.

La macro `NULL` se encuentra definida en el achivo de cabecera `stdio.h`.

El programa se auxilia de tres funciones:

a) La función `borrar()` elimina el último nodo de la lista
b) La función `recorre()` sirve para posicionarse en el último nodo de la lista
c) La función `lectura()` obtiene una cadena de la unidad de entrada

El programa declara una variable llamada renglon que funge como cabeza de lista. Su dirección es almacenada en la variable `primero` ya que es el primer nodo de la lista y siempre que se recorra la lista se empezara por él. Este nodo no puede ser borrado. No contiene información y apunta al primer elemento con información. Al inicio de las operaciones apunta al valor de nulo que llamaremos *tierra*.

El programa despliega el caractér < > >. Dicho caractér funcionará como un prompt de comandos que indicará al usuario cuando pueda introducir alguna orden.

La opción de inserción permitira al usuario introducir tantas lineas como se desee. La inserción siempre se realiza en la última linea del texto. La variable nodo siempre apunta al nodo recien insertado. La inserción se realiza obteniendo memoria para un nuevo nodo. Si la petición resulta satisfactoria entonces el último nodo debe apuntar a la dirección de memoria recien obtenida.

Para salir de la opción de inserción bastará con pulsar la tecla de ESCAPE. Cuando se pulsa la tecla ESCAPE es necesario eliminar el nodo generado en la inserción ya que dicho nodo no contiene información. Este problema puede ser solucionado agregando una variable auxiliar para leer el texto antes de realizar la petición de memoria para el nuevo elemento. Queda al lector la realización de dicha modificación.

Cuando una linea del texto se debe eliminar, el nodo ya no contiene información y por lo tanto podemos liberar la memoria que ocupa. Una vez que dicha memoria es devuelta al sistema, la información se pierde de manera definitiva y no hay manera de recuperarla.

El listado de los nodos no se realiza de manera secuencial, como había sucedido hasta este momento. Cada nodo es accesado preguntando por el miembro siguiente en donde está almacenada la dirección del próximo nodo. Este a su vez contiene en el mismo miembro la dirección del próximo nodo. Esto se repite hasta que se detecta el nodo que contiene la tierra. Dicho nodo siempre es el último de la lista

Pseudocódigo:

RECORRE

```
APUNTA AL PRIMER NODO
HAZ MIENTRAS (EXISTA UN NODO SIGUIENTE)
      INDICA QUE EL NODO ACTUAL SE CONVERTIRA EN EL NODO ANTERIOR
      INDICA QUE EL NODO SIGUIENTE ES AHORA EL NODO ACTUAL
FIN DEL HAZ
```

Codigo:

```
/* Recorre la lista hasta el ultimo elemento */
void recorre (void)
{
      nodo = primero;
      anterior = primero;
      while ( nodo->siguiente)
      {       anterior = nodo;
            nodo = nodo->siguiente;
      }
}

/* Esta funcion borra el ultimo nodo insertado */
void borrar(void)
{
      free(nodo);
      nodo = anterior;
      if (nodo)
            nodo->siguiente = NULL;
}


/* Lee el texto de la pantalla.*/
void lectura(void)
{
      int contador=0;

      /* MIentras no se alcance el fin de la linea */
      while (contador < MAXLONG)
      {
            /* Obtiene un caracter sin desplegarlo */
            nodo->texto[contador]= getch();

            switch (nodo->texto[contador])
            {

            /* Salida */
            case ESC:
                  nodo->texto[0] = ESC;
                  contador = MAXLONG + 1;
                  break;

            /* Retorno de carro */
            case ENTER:
                  nodo->texto[contador] = '\0';
                  contador = MAXLONG + 1;
                  break;

            /* Tecla BACKSPACE */
            case REGRESO:
                  printf("\b \b");
                  contador=contador-2;
                  break;
```

```
            /* Cualquier otro caracter */
            default:
                  printf("%c",nodo->texto[contador]);
                  break;
            }
            contador++;
      }
}
```

Resultados:

```
 ?: Ayuda
>
 I Inserta texto
 B Borra ultima linea de texto
 L Lista todo el texto
 ESC Salida
>I
        :Ejemplo del nuevo procesador de textos
        :desarrollado en lenguaje C
        :      Este es un ejemplo de lo que se puede hacer
        :
        :con este programa
        :
>L

        Ejemplo del nuevo procesador de textos
        desarrollado en lenguaje C
                Este es un ejemplo de lo que se puede hacer

        con este programa
>B
         con este programa

                 Desea borrar? [S/N] s
>I
        :con un poco de imaginación
        :
>L

        Ejemplo del nuevo procesador de textos
        desarrollado en lenguaje C
                Este es un ejemplo de lo que se puede hacer

        con un poco de imaginación
> <ESC>
```

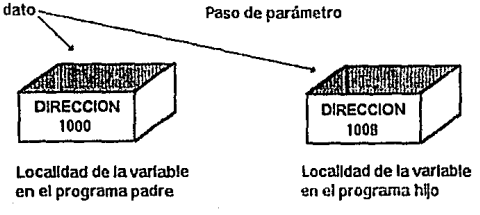
II.4.4 Apuntadores como argumentos en funciones

El paso de parámetros por valor es muy útil cuando deseamos escribir funciones que manipulen los parámetros sin preocuparnos los cambios que estos puedan tener. Debido a que los valores se almacenan en localidades distintas de memoria los cambios no se ven reflejados en las variables que fueron usadas como parámetros. Por otra parte una función es capaz de entregar un valor como código de retorno al módulo padre que lo ha mandado llamar.

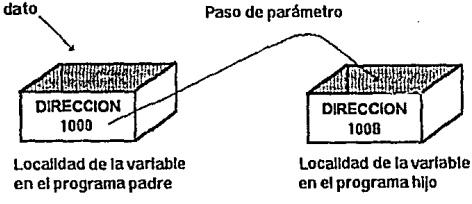
En más de un caso esto no es suficiente. Un módulo que lea más de un valor no puede regresar dichos valores con una sentencia `return()`. Por ejemplo, para modularizar el programa con el que realizamos la suma de dos matrices es necesario que la matriz de resultado sea modificada por la función.

Si se llama a una función y se desea que los cambios en los valores de los parámetros se reflejen en el módulo que realiza la llamada no se debe de mandar el valor de la variable sino la **dirección** que dicha variable ocupa en memoria. Tal como el nombre lo indica en el paso de parámetros por valor las variables copian el **valor** en las variables del módulo hijo y ocupan **direcciones distintas** de memoria, cuando se pasan parámetros por referencia se copia la **dirección** de memoria en la variables del módulo hijo. El nombre de esta técnica hace alusión a la referencia a la dirección que se emplea en el módulo hijo.

Para clarificar la diferencia entre estos dos métodos observemos el gráfico.



Paso de parámetros por valor



Paso de parámetros por referencia

En los gráficos mostrados se tiene la localidad 1000 que suponemos pertenece a una variable del programa principal. Si se realiza un llamada con las formas expuestas hasta este momento, la copia del valor del dato es pasada hacia la localidad de memoria de la variable en el módulo hijo . Al pasar dicho valor se rompe cualquier vínculo con la variable de donde fue extraída.

En el paso de parámetros por referencia se copia la dirección dentro de la variable del módulo hijo y por tanto el vínculo con dicha variable permanece. Por supuesto, la localidad 1008 que antes aceptaba un tipo de dato especifico ahora tiene que acepta una *dirección de memoria a un dato específico.*

La especificación del paso de parámetros por referencia se hace utilizando los operadores * y & los cuales son complementarios. El ampersand lo hemos utilizado con gran frecuencia a lo largo de estos apuntes. Cada vez que hemos utilizado una *función* **scanf()** con una variable escalar, el operador & ha estado presente. Veamos lo que esto significa. Cuando anteponemos el operador ampersand a un nombre de variable estamos obteniendo la dirección de memoria en donde se encuentra localizada. De tal forma que la *función* **scanf()** utiliza la dirección de la variable en la que colocará el valor obtenido del dispositivo de entrada. Si nosotros omitimos el ampersand para una variable escalar, estaremos enviando un valor que **scanf()** interpretará como una dirección. Dicho valor puede ser cualquiera dentro de la RAM y podría apuntar a una dirección de vital importancia. Esta dirección podría ser una sección del sistema operativo.

Observemos el siguiente programa:

Programa :
Hacer un programa que lea una fecha desde el teclado en forma numérica y la imprima en su forma larga (con letras).

Solución:

```
/* Ejemplo 37: Paso de parametros por referencia
   25/feb/93
   Ejemplo: LEE FECHA */

/* Declaracion de funciones prototipo  */
void leefecha(int *dia, int *mes, int *anno);
void pintafecha(int dia, int mes, int anno) ;

/* Inicia el programa principal */
void main(void)
{
   /* Declaracion de variables locales   */
   int dia,mes,anno;

      clrscr();
      printf("\n\tLectura de una fecha ");
      printf("\n\t=========================\n\n");

      leefecha(&dia, &mes, &anno);
      printf("\n La fecha es:");
      pintafecha(dia, mes, anno);
}
```

Observaciones al programa principal:

En el programa principal se utilizan dos funciones, **leefecha()** y **pintafecha()** . La función **pintafecha()** está destinada a desplegar la fecha según el *formato largo* especificado en el enunciado del problema. La función acepta parámetros por valor y no tiene nada de nuevo para nosotros. La función **leefecha()** tiene una función prototipo en la cual cada variable tiene antepuesto un asterisco. Esta es la indicación de que se trata de punteros a variables de tipo entero.

Dentro de **main()** de realiza la llamada a la función **leefecha()**, observamos que se antepone un ampersand a cada variable para enviar la dirección de dichas variables y no su valor.

Pseudocódigo:

```
LEE FECHA(dia, mes, anno)

REPITE
      LEE(dia)
      SI(dia_no_valido)
            ESCRIBE(" Revise que el dia sea correcto")
      FIN DEL SI
HASTA(que el dia sea valido)

REPITE
      LEE(mes)
      SI(mes_no_valido)
            ESCRIBE(" El mes digitado es incorrecto")
      FIN DEL SI
HASTA (que el mes sea válido)

REPITE
      LEE(año)
      SI(año_no_válido)
            ESCRIBE(" El año digitado es incorrecto")
      FIN DEL SI
HASTA(que el año sea válido)
```

Código:

```
/* Funcion leefecha   */
void leefecha(int *dia,int *mes, int *anno)
{
int no_valido;
printf("\n FECHA");
```

```
/* Lectura de la fecha */
do
{
      printf("\n \t Dia:[1-31] ");
      scanf("%d",dia);
      no_valido = (*dia < 0) || (*dia > 31);
      if (no_valido)
            printf ("\n\a Revise que el dia sea correcto");
} while (no_valido);

do
{
      printf( "\t Mes:[1-12] ");
      scanf("%d",mes);
      no_valido = *mes <= 0 || *mes > 12;
      if (no_valido)
            printf ("\n\a El mes digitado es incorrecto \n\a");
} while (no_valido);
do
{
      printf( "\t Año:[1900-3000] ");
      scanf("%d",anno);
      no_valido = *anno < 1900 || *anno > 3000;
      if (no_valido)
            printf ("\n\a El año digitado es incorrecto \n\a");
} while (no_valido);

}

/* Despliega la fecha */
void pintafecha(int dia, int mes, int anno)
{
const char nombre_mes[12][11]=
      {"ENERO","FEBRERO","MARZO","ABRIL","MAYO","JUNIO",
       "JULIO","AGOSTO","SEPTIEMBRE","OCTUBRE","NOVIEMBRE",
       "DICIEMBRE"};

      printf ("%2d de %s de %4d",dia,nombre_mes[mes-1],anno);
}
```

Observaciones:

La función **leefecha()** declara los parámetros anteponiendoles un asterisco, esto convierte en punteros a las variables así declaradas.

Dado que estas variables ya contienen direcciones no es necesario colocar el amperasand en las lecturas con el **scanf()**, solamente el nombre de la variable. En cambio cada vez que se desea manipular el valor almacenado en dicha dirección se utiliza un asterisco.

VISIBILIDAD

Las llaves izquierda y derecha delimitan en cualquier estructura un ámbito de control. Cualquier cosa encerrada entre un par de llaves es accesible a la estructura o superestructura que abre las llaves. Pero no es

posible accesarlas por una estructura fuera de ellas. Las sentencias que conforman el bloque verdadero de un if solo pueden ser accesadas por una condición verdadera en el if que las abarca. O bien, no es posible dar un salto al interior de un ciclo for si no es entrando directamente por el for que controla dichas sentencias.

Las variables no escapan a esta regla. Si una variable es declarada dentro de un par de llaves, sólo son accesibles por las sentencias que se encuentren encerradas por el mismo par de llaves. Así cualquier variable declarada después de la llave izquierda de la función `main()` solo podrá ser accesada por las sentencias de `main()`, y no por las sentencias que conformen las funciones que sean llamadas por `main()` aunque estas se encuentren codificadas en el mismo archivo fuente.

Si una variable se encuentra declarada antes de la función `main()`, entonces dicha variable podrá ser utilizada por cualquier sentencia que se encuentre en el mismo archivo.

La *visibilidad* de una variable queda definida por su capacidad para ser accesada u ocultada del resto de las funciones que conforman un programa. Si una variable solo puede ser vista por la función que la declaró entonces la variable recibe el nombre de *variable local*. Si una variable es declarada para ser vista por la función que la genera y por todas las rutinas que esta llame, o por todas las funciones que se encuentren en el mismo archivo, esta variable recibe el nombre de *varible global*. Por definición, cualquier variable es local para el conjunto de llaves entre las que ha sido declarada.

Según se estableció en el capítulo uno, es muy deseable que las variables con las cuales se comunican los módulos, se pasen a las funciones por medio de la lista de parámetros que definen a la función. Es decir, lo más recomendable en un sistema es que la gran mayoría de las variables sean de tipo local. Las variables globales deben ser evitadas tanto como la situación lo permita.

Una de las principales razones por las cuales los programadores suelen usar variables globales en lugar de sus contrapartes locales, es que en un sistema típico, suelen manejarse los mismo datos, y las misma variables a lo largo de un proceso. Por ejemplo, en una nómina, los nombres de los empleados, y sus datos necesarios para cálculo (horas trabajadas, faltas, vacaciones, permisos, horas extras, etc) son utilizados en la mayor parte de las funciones del sistema. Programar cada función utilizando largas listas de parámetros que difieran solo en una o dos variables, hace al trabajo rutinario y monótono... Pero según se ha mencionado, también lo vuelve más eficiente. Aquí el verdadero problema no es que la lista de parámetros se repita de una función a otra en forma prácticamente idéntica sino que las bases de datos del sistema no están siendo manejadas de una forma correcta o incluso no fueron diseñadas de un modo eficiente para lograr un flujo ágil de la información a través del sistema.

¿Y que hay de las variables que no pueden ser almacenadas en una simple base de datos? Se preguntan muchos programadores. Es necesario recordar que los módulos deben ser diseñados de tal forma que deban ser tan independientes como sea posible del algoritmo con el cual se ha implantado el sistema. Pasar como parámetros a variables como `salto_hoja`, `pagina`, `contador`, `bandera`, `finproceso` hacen a la funcion dependiente de variables específicas, generadas por un superordinado específico, que se ejecuta bajo un algoritmo específico. La flexibilidad para cambiar el algoritmo o un simple enfoque desaparece. Por ejemplo, la variable `contador` puede pertenecer a un apuntador a un registro de procesamiento de datos secuencial. ¿ La variable `contador` posee el número del siguiente registro a procesar o el del último que se procesó? Algo tan simple como esto puede acarrear múltiples problemas en cualquier sistema en funcionamiento y mayores para los programadores que se encuentren depurando módulos que no fueron hechos por ellos mismos.

Los fabricantes de software siguen produciendo compiladores con variables globales y con variables locales. Los libros de programación (y muchos profesionistas) continuan utilizando sin miramientos las variables globales. Definitivamente, la idea no es condenar a la hoguera y convertir la filosofía de la programación en una inquisición. Si los compiladores siguen saliendo de fábrica con estas características es porque siguen siendo necesarias. Por un lado aún no se logra una estandarización en cuanto a los métodos y procedimientos que la producción de software requiere. La Programación estructurada y el Diseño

estructurado no son las únicas corrientes de pensamiento que existen en el mercado de trabajo. Con mayor tristeza es necesario apuntar que muchas personas ni siquiera tienen idea de que existen dichas filosofías, o pero aún, no conocen qu existe alguna técnica o escuela de programación.

La segunda razón (más práctica) para que los compiladores continúen en esta tradición, es la compatibilidad con versiones anteriores.

Existe más de una caso en el cual una variable global es definitivamente más útil y eficaz que una local. Un arreglo cuyo contenido son los nombre de los meses es utilizado en la mayor parte de los módulos que despliegan o imprimen la fecha. Es más sencilla declararla como una variable global y no pasarla como parámetro a los módulos que la requieren. Un arreglo, o una estructura conteniendo los colores que la pantalla desplegará caé en el mismo caso. O el ejemplo visto anteriormente en el cual el número máximo que puede contener una variable de tipo entero no se va a modificar a lo largo del programa.

Los ejemplo presentados tienen una característica común que es muy importante para poder considerar la posibilidad de utilizarlas en forma global: son *variables* que no cambian. El modo de pantalla se establece al entrar en funcionamiento el sistema y una vez verificada la posibilidad cromática y de resolución esta no va a alterarse a lo largo de la sesión de trabajo. Los meses del año tampoco varían porque el sistema se ejecute en alguna estación del año en particular o porque se utilicen plataformas de trabajo diferentes (Unix. DOS, Macintosh, etc)

Ejemplo:
Hacer un programa que calcule la fecha del día de mañana en base al día de hoy.

Solución:
Aunque el programa es largo, se utilizan los programas que se escribieron para la lectura de la fecha en un ejemplo precedente. Los programas que ya se habían escrito solo han recibido muy pequeños cambios.

En cambio el proceso de validación ha sido ampliado con la creación de un rutina especificamente diseñada con ese fin.

Pseudocódigo:

```
FECHA MAÑANA

   DECLARA VARIABLES
   MIENTRAS (se desee continuar)
      ESCRIBE("Cálculo del dia de mañana ")
      ASIGNA A no_valido EL CODIGO DE RETORNO DE leefecha(dia, mes, anno);
      HAZ CASO DE(no_valido)
            caso 0:
                  CALCULA LA FECHA DE MAÑANA
                  DESPLIEGA LA FECHA DE HOY
            case 1:
                  ESCRIBE(" El dia digitado es incorrecto")
            case 2:
                  ESCRIBE("  El mes es incorrecto")

            POR OMISION:
                  CONTINUA
      FIN DEL CASO
      ESCRIBE("Pulse una tecla para continuar... <ESC>: SALIDA")
      LEE UN CARACTER
  FIN DEL HAZ MIENTRAS
```

Solución:

```
/* Ejemplo 36 */
/* 3/Dic/92 */
/* Ejemplo: EL DIA DE MAÑANA */

/* Declaracion de funciones prototipo  */
int leefecha(int *dia, int *mes, int *anno);
int verifecha ( int dia,int mes, int anno);
int mannana (int dia, int mes, int anno);
int mod (int divisor, int dividendo);
void pintafecha(int dia,int mes,int anno);

/* Variables globales */
const int diames[12]=
      {31,28,31,30,31,30,31,31,30,31,30,31};
const char nombre_mes[12][11]=
      {"ENERO","FEBRERO","MARZO","ABRIL","MAYO","JUNIO",
       "JULIO","AGOSTO","SEPTIEMBRE","OCTUBRE","NOVIEMBRE",
       "DICIEMBRE"};

/* Inicia el programa principal */
void main(void)
{
   /* Declaracion de variables locales   */
   int dia,mes,anno;
   int no_valido;
   char opcion;
   const int ESC=27;

   /* Ciclo de control del programa */
   opcion = '';
   while (opcion != ESC )
   {
      clrscr();
      printf("\n\tCálculo del dia de mañana ");
      printf("\n\t==========================\n\n");

      no_valido = leefecha(&dia, &mes, &anno);

      /* Despliega el mensaje de error en base al código retornado
      por leefecha                                                */
      switch (no_valido)
      {
            case 0:
                  mannana (dia, mes, anno);
                  printf ("\n La fecha de hoy es :");
                  pintafecha(dia,mes,anno);
                  break;
            case 1:
                  printf(" \n La fecha no es consistente");
                  printf(" \n\a   El dia digitado es incorrecto\n");
                  break;
```

```
            case 2:
                  printf(" \n\a La fecha no es consistente");
                  printf(" \n El mes es incorrecto\n");
                  break;
            default:
                  break;
      }

      printf("\n\n Pulse una tecla para continuar... <ESC>: SALIDA");
      opcion = getch();


   }
}
```

Observaciones al programa principal :

En este programa la función `leefecha()` devuelve un código de retorno que será empleado para desplegar un mensaje que diagnostique el error que se detecte durante la lectura de la fecha. En caso de que el código de retorno sea cero (sin error), se procederá a calcular la fecha del día siguiente.

Pseudocódigo:

```
leefecha(dia, mes, anno)
/* Lectura de la fecha */

REPITE
      LEE(dia)
      SI(dia_no_válido)
            ESCRIBE(" Revise que el dia sea correcto")
      FIN DEL SI
HASTA(que el dia sea válido)

REPITE
      LEE(mes)
      SI(mes_no_válido)
            ESCRIBE(" El mes digitado es incorrecto")
      FIN DEL SI
HASTA(que el mes sea válido)do

REPITE
      LEE(año)
      SI(año_no_válido)
            ESCRIBE(" El año digitado es incorrecto")
      FIN DEL SI
HASTA(que el año sea válido)

ASIGNA A no_válido EL CODIGO DE verificacion_de_fecha(dia,mes,anno)
RETORNA EL VALOR DE no_válido)
```

```
/* Funcion leefecha
   Devuelve
      0 si la fecha es correcta
   */
int leefecha(int *dia,int *mes, int *anno)
{
int no_valido;

printf("\n FECHA");

/* Lectura de la fecha */
do
{
      printf("\n \t Dia:[1-31] ");
      scanf("%d",dia);
      no_valido = (*dia < 0) || (*dia > 31);
      if (no_valido)
            printf ("\n\a Revise que el dia sea correcto");
} while (no_valido);

do
{
      printf( "\t Mes:[1-12] ");
      scanf("%d",mes);
      no_valido = *mes <= 0 || *mes > 12;
      if (no_valido)
            printf ("\n\a El mes digitado es incorrecto \n\a");
} while (no_valido);
do
{
      printf( "\t Año:[1900-3000] ");
      scanf("%d",anno);
      no_valido = *anno < 1900 || *anno > 3000;
      if (no_valido)
            printf ("\n\a El año digitado es incorrecto \n\a");
} while (no_valido);

no_valido = verifecha(*dia,*mes,*anno);
return (no_valido);
}
```

```
VERIFICACION DE LA FECHA (dia, mes, anno)

APAGA BANDERA DE no_válido // CODIGO DE RETORNO
SI(dia > 0) ENTONCES

      HAZ CASO DE (mes)
            caso de un mes de 31 dias:
                  SI (dia >31 ) ENTONCES
                     ENCIENDE BANDERA no_válido
                  FIN DEL SI
```

```
            caso de mes de 30:
                  SI (dia > 30)ENTONCES
                     ENCIENDE BANDERA no_válido
                  FIN DEL SI
            caso de mes de Febrero :
                  SI (dia > 28 ) ENTONCES
                     ENCIENDE BANDERA no_válido
                  FIN DEL SI
                  SI (ES DIA 29 y año bisiento) ENTONCES
                     APAGA LA BANDERA   // LA FECHA ES CORRECTA
                  FIN DEL SI
            por omision:
                  ASIGNA mes_no_válido AL CODIGO DE RETORNO
      FIN DEL CASO
SINO
   ASIGNA dia_no_válido AL CODIGO DE RETORNO
FIN DEL SI
RETORNA (no_válido como CODIGO DE RETORNO)


 /*  Rutina verifecha.
     Verifica que una fecha determinada sea correcta
 Esta rutina devuelve:

      0 si la fecha es correcta
      1 si el dia es incorrecto
      2 si el mes es incorrecto
 */

int verifecha ( int dia,int mes, int anno)
{
int no_valido;

no_valido = 0;
if (dia > 0)
{
      switch (mes)
      {
            /* Meses de 31 dias */
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                  if (dia >31 )
                     no_valido = 1;
                  break;
            /* Meses de 30 dias */
            case 4:
            case 6:
            case 9:
```

```
            case 11:
                  if (dia > 30)
                     no_valido = 1;
                  break;
            /* Febrero                 */
            case 2:
                  if (dia > 28 )
                      no_valido = 1;

                  /* Si es 29 y año bisiento la fecha es correcta */
                  if (!mod(anno, 4) && (dia == 29))
                      no_valido = 0;

                  break;
            default:
                  no_valido = 2;
      }
}
else
   no_valido = 1;
return (no_valido);
}


mannana (dia, mes, anno)

APAGA BANDERA no_válido
SI (dia > 0) ENTONCES
      HAZ CASO DE mes)
            caso de los meses de 31 dias
                  SI( dia es ultimo de mes) ENTONCES
                        ASIGNA QUE ES dia PRIMERO DE MES
                        SI (mes es diciembre ) ENTONCES
                              ASIGNA QUE NUEVO mes ES ENERO
                              INCREMENTA EL año
                        SINO
                              INCREMENTA mes
                        FIN DEL SI
                  SINO
                        INCREMENTA dia
                  FIN DEL SI
            caso de los  meses de 30 dias:
                  SI(dia es ultimo de mes) ENTONCES
                        ASIGNA QUE ES dia PRIMERO DE MES
                        INCREMENTA mes
                  SINO
                        INCREMENTA dia
                  FIN DEL SI
```

```
            caso del mes de Febrero:
                  SI (dia es ultimo de mes) ENTONCES
                        SI ( dia es 28 y año bisiento ) ENTONCES
                              INCREMENTA EL dia
                        SINO
                              ASIGNA QUE ES dia PRIMERO DE MES
                              INCREMENTA mes

                        FIN DEL SI
                  SINO
                        INCREMENTA dia
                  FIN DEL SI
            por omision:
                  ASIGNA fecha_no_valida AL CODIGO DE RETORNO
      FIN DEL CASO
      SI ( fecha valida ) ENTONCES
            ESCRIBE("La fecha de mañana es: ")
            pintafecha(dia,mes,anno)
      FIN DEL SI
SINO
      ASIGNA fecha_no_valida AL CODIGO DE RETORNO
FIN DEL SI
RETORNA (no_válido)


/* Rutina mannana
    Calcula la fecha de mañana */


int mannana (int dia, int mes, int anno)
{
int no_valido;

no_valido = 0;
if (dia > 0)
{
      switch (mes)
      {
            /* Meses de 31 dias */
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
            /* Meses de 31 dias */
                  /* Si es ultimo de mes */
                  if (dia >30 )
                  {
                        dia = 1;
```

```
                        /* Si es diciembre */
                        if (mes == 12)
                        {
                              mes =1;
                              anno++;
                        }
                        else
                              mes++;
                  }
                  else
                        dia++;
                  break;
            /* meses de 30 dias */
            case 4:
            case 6:
            case 9:
            case 11:
                  if (dia >29)
                  {
                        dia = 1;
                        mes++;
                  }
                  else
                        dia++;

                  break;
            /* Febrero              */
            case 2:
                  if (dia > 27 )
                  {
                        /* Si es 28 y bisiento mañana es 29 */
                        if (!mod(anno,4) && dia == 28)
                              dia++;
                        else
                        /* sino es dia primero */
                        {
                              dia = 1;
                              mes++;
                        }
                  }
                  else
                        dia++;
                  break;
            default:
                  no_valido = 1;
      }
      if (!no_valido)
      {
         printf("La fecha de mañana es: ");
         pintafecha(dia,mes,anno);
      }
}
```

```
else
      no_valido = 1;
return (no_valido);
}

/* Obtiene el residuo de una division */

int mod( int dividendo, int divisor)
{
  return (dividendo-(dividendo/divisor)*divisor);
  }
/* Despliega la fecha */
void pintafecha(int dia, int mes, int anno)
{
      printf ("%2d de %s de %4d",dia,nombre_mes[mes-1],anno);
}
```

II.4.6 Operaciones sobre apuntadores

La aritmética de los apuntadores es básicamente la misma que sobre cualquier otro tipo de dato. Salvo por el caso que se realiza en base a numeros de bytes.

Un puntero se declara para un tipo de dato específico. Aunque una dirección siempre tiene un formato específico independientemente del dato que se almacene. Sin embargo para evitar errores que pueden ser detectados por el compilador, se obliga al programador a respetar las convenciones que realizaría el compilador si él trabajara directamente sobre los datos.

Hemos visto que un arreglo puede implantarse con un puntero al cual se le *suman el número de elementos* que existían entre el primer elemento y el que deseemos accesar. Un tipo entero suele requerir de dos bytes para representarlo. Si un puntero entero tiene la localidad 1000 y se le realiza la siguiente sentencia

```
int *puntero;
puntero = puntero + 2;
```

Al final de la sentencia el valor del puntero será de 1004. Por otra parte los operadores de incremento y decremento desplazan hacia adelante o hacia atras respectivamente. El desplazamiento se realiza en base a elementos.

Por ejemplo, para una estructura que requiera de 16 bytes definimos un puntero de la siguiente manera:

```
struct item undato;
struct item *identificador;

identificador=&undato;
++identificador;
```

Si `undato` está localizado en la dirección 1000, el valor almacenado en identificador tras la operación incremento será 1017.

Las operaciones válidas solo son la suma o resta de valores enteros. No tiene sentido hablar de multiplicaciones o divisiones de direcciones, así como tampoco lo tiene el trabajar con sumas o restas de datos con parte fraccionaria.

Por otra parte, aunque es posible comparar la magnitud de dos punteros, no tiene ningún caso basar código en una comparación de ese tipo. Cada vez que un programa es ejecutado, es cargado en localidades distintas de memoria. No existen posiciones absolutas y no se puede asegurar que una petición de memoria entrege un valor superior a una petición que le anteceda en el tiempo.

II.5. Archivos

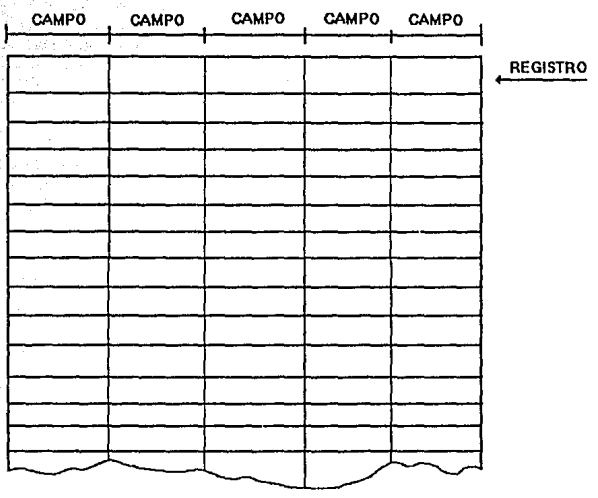
La mayoria de las aplicaciones que se utilizan y se escriben en la actualidad requieren que se grabe la información en un medio permanente. El editor de texto que escribimos en un ejemplo precedente no tiene una gran utilidad si no es capaz de grabar la informacioón en un archivo. Este apartado esta plenamente dedicado al manejo de los archivos que contendrán información.

II.5.1 Como abrir y cerrar un archivo de datos

Un archivo es una coleccion de datos relacionados entre si que se encuentran almacenados en un medio secundario. Dicho medio puede ser una unidad de disco, una cinta, etc.

Cabe hacer notar que en el sistema UNIX y en C++ un archivo puede ser cualquier objeto al cual se pueda accesar con operaciones de entrada y/o salida. Un terminal de video, un teclado o la impresora pueden ser tratados como archivos. En dichos sistemas, la comunicación con los archivos se realiza a través de *corrientes de datos* o *secuencias (streams)*. Las secuencias proporcionan una interface independiente de los detalles de más bajo nivel, de los cuales se hará cargo el sistema operativo.

La forma más común de representar a un archivo consiste de una tabla conformada por renglones y columnas. Cada columna recibe el nombre de *campo* y cada renglón recibe el nombre de *registro*.



Un arreglo de estructuras puede ser el mejor ejemplo de lo que esta idea significa en forma práctica. Cada miembro de una estructura es lo que nosotros hemos llamado *campo* de información y cada estructura que conforma el arreglo, constituye un *registro* de información

Un archivo es independiente de cualquier programa. Si un programa requiere accesar a un archivo, debe realizar una petición al sistema operativo. Dependiendo de la implantación se crean áreas especiales de trabajo para que el programa pueda trabajar con los datos del archivo.

Cuando un programa desea trabajar con un archivo, primero debe *abrir* el archivo. La apertura de un archivo establece la comunicación necesaria para que un programa pueda accesar los datos. Un archivo puede estar escrito de dos formas distintas: binaria y texto.

Un archivo binario almacena la información tal y como la representa en forma interna. El número de caracteres escritos en igual al número de caractéres desplegados en un listado de directorio. Básicamente estos archivos almacenan la información de una forma más conveniente para el programador.

Un archivo texto es aquel que almacena caracteres. Generalmente se organiza en renglones y cada renglón posee un salto de linea. Un archivo de texto es legible directamente por una persona cuando se lista en la pantalla su contenido. Este tipo de archivos realizan una serie de conversiones cuando son desplegados. Por ejemplo, se puede almacenar un caractér `'\t'` que ocupa una sola posición caractér pero ser desplegado como una salto de tabulador.

Un archivo puede ser abierto en diversos modos, los cuales son listados a continuación:

a) Modo r :	En este modo se abre un archivo para lectura. El archivo debe exitir previamente
b) Modo w:	Un archivo es creado para ser utilizado solo en escritura, si el archivo existe su contenido es destruido
c) Modo a:	Un archivo es abierto para añadir al final del mismo.
d) Modo rb:	Abre un archivo binario para solo lectura
e) Modo wb	Crea un archivo binario para escritura
f) Modo ab	Añade información a un archivo binario
g) Modo r+	Abre un archivo existente para operaciones de lectura/escritura
h) Modo w+	Crea un archivo para operaciones de lectura/escritura
i) Modo a+:	Añade o crea un archivo para operaciones de lectura/escritura
j) Modo rb+:	Abre un archivo binario para operaciones de lectura/escritura
k) Modo wb+:	Crea un archivo binario para operaciones de lectura/escritura
l) Modo ab+:	Añade información a un archivo binario para operaciones de lectura/escritura

Modos de apertura de un archivo

Cuando un archivo es abierto, se espera que este exista, de no ser así se intentará crearlo. En el modo w cualquier información previa existente en el archivo es borrada. Como se puede apreciar, si no se especifica ninguna otra cosa, el archivo abierto es de tipo texto.

La función que realiza la apertura de un archivo es **`fopen()`** y tiene la siguiente forma general:

*FILE *fopen(const char *nombre_archivo, const char *modo) ;*

Donde:

FILE	Es un tipo especificado en en el archivo de cabecera stdio.h
nombre_archivo	Es el nombre del archivo que se debe abrir
modo	Es alguno de los modos descritos anteriormente

Ejemplo:

Para abrir un nuevo archivo llamado alumnos en el cual solo se realizará escritura de texto se deben escribir las siguientes sentencias:

```
FILE *arch
arch = fopen ("alumnos","w");
```

Para realizar una apertura en la cual se desee leer el contenido se tendrá:

```
arch= fopen("alumnos", "r");
```

Y para continuar añadiendo información se tendrá:

```
arch = fopen("alumnos","a");
```

Cuando un archivo ya no será necesitado por un programa, es muy recomendable que el archivo sea *cerrado*, es decir, que se elimine la conexión entre el programa y el archivo. La función **`fclose()`** cumple con dicha tarea. Su función prototipo es la siguiente:

*Int fclose(FILE *puntero_archivo);*

Donde:

`puntero_archivo` Corresponde al puntero del archivo que se desea cerrar

II.5.2 Creación de un archivo de datos

Existen diversas funciones que nos permiten escribir o leer de un archivo que ha sido abierto con anterioridad.

La función `putc()` y la funcion `fputc()` son equivalentes. Con ellas se puede escribir un caractér en un archivo. Su función prototipo es:

*Int fputc(int caracter, FILE *puntero_archivo)*

Donde:

caracter Es el caractér que se va a escribir en el archivo.
puntero_archivo Es el puntero a un archivo abierto

El caractér que se escribirá está definido como de tipo entero en la función prototipo, sin embargo `fputc()` solo escribirá la parte menos significativa. Esto es importante porque hemos estado utilizando con bastante libertad la función `getche()` la cual entrega un valor entero, que para ciertas implantaciones puede ocupar todo el entero en las teclas especiales. Un retorno de carro que ha sido leído con `getche()` puede no ser interpretado correctamente por `fputc()` ya que parte del valor es cortado al ser escrito en el archivo.

Ejemplo:

Hacer un programa que lea una cadena y posteriormente la escriba en un archivo

```
/* EJEMPLO 521.C
Uso de arcchivos */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef SALIDA
      #define SALIDA '@'
#endif

void main(void)
{
FILE *arch;
char linea[80];
char archivo[12];
register int contador;

/* Pide el nombre del archivo a crear */
clrscr();
printf("\n Digite el nombre del archivo: ");
gets(archivo);
```

```
if (strlen(archivo) == 0)
{
      printf("\n Se omitio nombre del archivo");
}
else
{
      /* Creacion del archivo */
      arch=fopen(archivo,"w");
      if ( arch != NULL )
      {

      /* Si la creacion fue exitosa captura una linea de texto a la vez */
      clrscr();
      linea[0]='\0';
      printf("\n @ Termina captura\n Capturando...\n");

      /* Mientras no se de caracter de salida */
      while (linea[0] != SALIDA)
      {

            gets(linea);
            contador=0;
            while (linea[contador] != '\0'&& linea[0] != SALIDA)
            {
                  /* Guarda un caracter a la vez */
                  fputc(linea[contador],arch);
                  contador++;
            }
            fputc('\n',arch);
      }
      /* Cierra el archivo */
       fclose(arch);
       }
      else
      {
            printf("\n No fue posible realizar la apertura");
      }

}
}
```

Observaciones:

Lo primero que solicita el programa es el nombre del archivo que será creado. Dicho nombre debe ser un nombre válido para el sistema operativo. Si el usuario no digita ningún nombre de archivo es programa no podrá ser accesado. En base a ese nombre se utiliza la instrucción `fopen()` para crear un archivo. Si el archivo ya existe toda la información que pueda contener será borrada.

Cuando el archivo ha sido correctamente abierto, se lee una cadena desde el dispositivo de entrada y dicha cadena es escrita caractér por caractér en el archivo. Debe recordarse que `gets()` no forma parte de la cadena, por lo que debemos introducirlo nosotros en el archivo, para que sea legible.

Cuando el usuario pulsa el caractér con el cual se termina la captura (una arroba), el archivo es cerrado y el programa finaliza.

LECTURA DE UN ARCHIVO

La lectura de un archivo se realiza con la función `fgetc()` y `getc()` que son idénticas. La forma general de dicha función es:

*int fgetc(FILE *puntero_archivo);*

La función prototipo entrega un entero como resultado de la lectura del caractér, pero solo la parte memos significativa posee un valor ya que la parte más significativa es cero. Cuando la función alcanza el fin de archivo envía una marca `EOF`. Tal como puede suponerse `EOF` es una macro definido dentro del archivo de cabecera `stdio.h`.

Una segunda función que es importante para realizar la lectura de una archivo es `feof()`, la cual inquiere por el fin del archivo. Su función prototipo es:

*int feof(FILE *puntero_archivo);*

En caso de que no se alcance el fin de archivo la función devuelve cero. Cuando se llega al final del archivo se obtiene el valor de cierto

Programa:
Modificar el ejemplo que crea el archivo para que realice la lectura del mismo.

```
/* EJEMPLO 522.C
Uso de archivos */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef SALIDA
      #define SALIDA '@'
#endif
#ifndef VERDADERO
      #define VERDADERO 1
#endif
#ifndef FALSO
      #define FALSO 0
#endif

void main(void)
{
FILE *arch;
char linea[80];
char archivo[12];
register int contador;

/* Pide el nombre del archivo a leer */
clrscr();
printf("\n Digite el nombre del archivo: ");
gets(archivo);
```

```
if (strlen(archivo) == 0)
{
      printf("\n Se omitio nombre del archivo");
}
else
{
      /* Apertura del archivo */
      arch=fopen(archivo,"r");
      if ( arch != NULL )
      {

      /* Si la peticion fue exitosa lee una linea de texto a la vez */
      clrscr();
      linea[0]='\0';
      printf("\n Listando el archivo...\n");

      /* Mientras no se detecte fin de archivo */
      while ( !feof(arch))
      {

            contador=0;
            linea[contador] =fgetc(arch);
            /* Recupera un caracter a la vez */
            while ( linea[contador] != '\n'&& !eof(arch))
            {
                  contador++;
                  linea[contador] =fgetc(arch);
            }
            linea[++contador]='\0';
            printf("%s", linea);
      }

      /* Cierra el archivo */
       fclose(arch);
       }
      else
      {
            printf("\n No fue posible realizar la apertura");
      }

}
}
```

Observaciones:

En general la estructura del programa no ha variado. Ahora se hace uso de la función **foef()** para controlar los ciclos del while. Si no se preguntara continuamente por el fin del archivo, el programa podría intentar accesar datos inválidos para el archivo provocando un error que seguramente terminaría por abortar la ejecución del programa.

Hemos optado por realizar un programa que lea una linea del archivo y entonces la despliege en la pantalla. A la linea de datos que procede del archivo es necesario agregarle el caracter nulo para que la cadena sea desplegada correctamente.

Programa:

Hacer un programa que:
a) Permita leer caracter por caracter un texto del dispositivo de entrada y almacenarlo en un archivo definido por el usuario.
b) Añadir a un archivo existente.
c) Lea cualquier archivo ASCII y lo despliege en pantalla.

Solución:

```
/* Uso de arcchivos */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef ESC
      #define ESC 27
#endif

int leenomarch(char archivo[]);
void captura(FILE *archivo , char salida);

void main(void)
{
FILE *arch;
char letra;
char archivo[12];
int opcion;

opcion='\0';
while (opcion != ESC)
{
      /* Menu */
      clrscr();
      printf("\n\t 1.- Crear archivo");
      printf("\n\t 2.- Leer el archivo");
      printf("\n\t 3.- Añadir a un archivo");
      printf("\n\t <ESC> Salida");
      printf("\n\n\n\t Digite su opcion: ");
      opcion = getche();
      switch(opcion)
      {
      /* Crea archivo */
      case '1':
            leenomarch(archivo);
            arch=fopen(archivo,"w");
            if ( arch != NULL )
            {
                  printf("\n Capturando...\n @ Termina captura\n");
                  captura(arch , '@');
                        fclose(arch);
            }
            else
```

```
            {
                  printf("\n No fue posible realizar la apertura");
            }
            break;

      /* Lee archivo existente */
      case '2':
            leenomarch(archivo);
            arch = fopen(archivo,"r");
            if  (arch != NULL)
            {
                  clrscr();
                  printf("\n");
                  letra = getc(arch);
                  while (letra != EOF )
                  {
                        putchar(letra);
                        letra = getc(arch);
                  }
                  fclose(arch);
            }
            else
            {
                  printf("\n No fue posible realizar la apertura");
            }
            getch();
            break;

      /* Añade a un archivo ya creado */
      case '3':
            leenomarch(archivo);
            arch=fopen(archivo,"a");
            if ( arch != NULL )
            {
                  printf("\n Capturando...\n @ Termina captura\n");
                  captura(arch , '@');
                   fclose(arch);
            }
            else
            {
                  printf("\n No fue posible realizar la apertura");
            }
            break;

      /* Salida */
      case 27:
            break;
      default:
            printf("\n Opcion invalida");
            break;

      }

}
}
```

```
/* leenomarch : lee nombre de archivo
   Lee un nombre de archivo
   Si no se pulsa un nombre se retorna cero*/

int leenomarch( char archivo[] )
{
int coderror = 0;

printf("\n Digite el nombre del archivo: ");
flushall();
gets(archivo);


if (strlen(archivo) == 0)
{
      printf("\n Se omitio nombre del archivo");
      coderror=1;
}
return(coderror);
}


/* Lee un caractér del dispositivo de entrada y lo almacena
en el archivo. La lectura continua hasta que se pulse el caracter
de salida que pasa como parámetro */

void captura(FILE *archivo , char salida)
{
char letra;

      letra = getchar();
      while (letra != salida)
      {
            puto(letra,archivo);
            letra=getchar();
      }
}
```

Observaciones:

Este programa escribe directamente del teclado al archivo. Cualquier tecla que sea pulsada provocará su inclusión. Esto incluye las teclas de retroceso que también serán grabadas en el archivo. En general es preferible usar un medio intermedio para poder editar los datos antes de que sean almacenados en un archivo.

Por otra parte aqui encontramos muy pocas sorpresas en la escritura del programa. Las opciones para crear un archivo y añadir a él se diferencian exlusivamente por el modo utilizado en la apertura.

Ejercicio:

Reescribir este programa con alguna de las opciones de apertura para que el código resulte más pequeño.

Aunque proporciona una gran flexibilidad, trabajar con caracteres individuales puede resultar un aspecto tedioso. En forma alterna se tiene las funciones **fputs() y fgets()**.

La función prototipo para **fputs()** es la siguiente:

*int fputs(const char *cadena, FILE *puntero_archivo)*

Donde:

**cadena* Es el puntero de la cadena que se va a ecribir en el archivo
**puntero_archivo* Corresponde al archivo en el cual se desea escribir

La función prototipo para **fgets()** es la siguiente:

*int *fgets(char *cadena, int longitud_de_cadena, FILE *puntero_archivo)*

Donde:

**cadena* Es el puntero de la cadena en donde se almacenarán los caracteres leidos en el archivo

longitud_de_cadena Es el número de caracteres que se leerán como parte de la cadena.

**puntero_archivo* Corresponde al archivo del cual se leerá la información

La función leerá *longitud_de_cadena-1* caracteres que se almacenarán en la cadena. Si se encuentra un salto de linea, se toma como fin de cadena. Si cualquiera de las funciones finaliza con error devolverán un valor de **EOF**.

Programa:
Modificar el ejemplo 4.5.1 reimplantandolo con la funcion fgets()

```
/* EJEMPLO 523.C
Uso de la funcion fputs()
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef SALIDA
    #define SALIDA '@'
#endif
#ifndef MAXLONG
    #define MAXLONG 80
#endif

void main(void)
{
FILE *arch;
char linea[MAXLONG];
char archivo[12];
register int contador;
```

```
/* Pide el nombre del archivo a crear */
clrscr();
printf("\n Digite el nombre del archivo: ");
gets(archivo);


if (strlen(archivo) == 0)
{
    printf("\n Se omitio nombre del archivo");
}
else
{
    /* Creacion del archivo */
    arch=fopen(archivo,"w");
    if ( arch != NULL )
    {

    /* Si la creacion fue exitosa captura una linea de texto a la vez */
    clrscr();
    linea[0]='\0';
    printf("\n @ Termina captura\n Capturando...\n");

    /* Mientras no se de caracter de salida */
    while (linea[0] != SALIDA)
    {

        gets(linea);
      contador=0;
      if (linea[contador] != SALIDA)
      {
            fputs(linea, arch);
            fputc('\n',arch);
      }
    }
    /* Cierra el archivo */
       fclose(arch);
       }
    else
    {
        printf("\n No fue posible realizar la apertura");
    }

}
}
```

Reescribir el ejemplo 4.5.2 reimplantandolo con la funcion `fgets()`

```
/* EJEMPLO 522.C
Uso de archivos */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#ifndef SALIDA
    #define SALIDA '@'
#endif
#ifndef MAXLONG
    #define MAXLONG 80
#endif

void main(void)
{
FILE *arch;
char linea[MAXLONG];
char archivo[12];

/* Pide el nombre del archivo a crear */
clrscr();
printf("\n Digite el nombre del archivo: ");
gets(archivo);


if (strlen(archivo) == 0)
{
    printf("\n Se omitio nombre del archivo");
}
else
{
    /* Apertura del archivo */
    arch=fopen(archivo,"r");
    if ( arch != NULL )
    {

    /* Si la creacion fue exitosa lee una linea de texto a la vez */
    clrscr();
    linea[0]='\0';
    printf("\n Listando el archivo...\n");

    /* Mientras no se detecte fin de archivo */
    while ( !feof(arch))
    {

       fgets( linea,MAXLONG, arch);
       if (!feof(arch))
         printf("%s", linea);
    }

    /* Cierra el archivo */
       fclose(arch);
       }
    else
    {
        printf("\n No fue posible realizar la apertura");
    }

}
}
```

II.5.3 Procesamiento de un archivo de datos

Un archivo de datos no solo es usado para almacenar en forma permanente la información contenidas en ellos. Esta información generalmente es procesada para obtener de ella resultados útiles.

Consideremos el caso de un sistema de control de alumnos para uso de un profesor. En un archivo se pueden almacenar los nombres y las calificaciones de cada uno de los alumnos, pero en general esta información es procesada para obtener las calificaciones definitivas de cada alumnos, establecer promedios por periodo, por alumnos e inclusive por grupo.

Un archivo de datos suele tener más de un tipo de dato. Para realizar una escritura a un archivo con datos que ocupan más de un byte se hace uso de la función `fwrite()` la cual tiene la siguiente función prototipo:

*size_t fwrite(void *buffer, size_t número_de_bytes, size_t cuenta, FILE *puntero_archivo)*

Donde:

size_t Es un tipo definido en stdio.h y es semejante al tipo unsigned

buffer Es un puntero a la localidad de memoria de donde serán escritos los datos hacia el archivo.

numero_de_bytes Número de bytes que se escribirán en el archivo

cuenta Es el número de elementos con *numero_de_bytes* de longitud que se escribirán en el archivo

puntero_archivo Corresponde al archivo en el cual se grabará la información.

La función retorna el número de elementos escritos en el archivo. Si la función terminó en forma correcta, entonces dicho valor será igual al de *cuenta*.

Ejemplo:

En un programa se trabaja con la siguiente estructura:

```
struct inf_alumno
{
char nombre[10];
char a_paterno[15];
char a_materno[15];
float examen[3];
float final[2];
} alumno[50];
```

Para almacenar el de estructuras en un archivo se escribiria el siguiente segmento de código:

```
#define NUMALUMNOS  25 /* Esta macro indica el numero de alumnos en el grupo*/
FILE *arch;
register int contador;
int correcto = 1;    /* bandera que indica si existió algun error en la escritura
                        al archivo */

/* Suponemos que en algun lugar se abre el archivo y almacena su puntero en ARCH*/

for (contador = 0 ; contador < NUMALUMNOS && correcto; contador++)
{
      correcto = fwrite(&alumno[contador],sizeof( struct inf_alumno),1,arch)
            if (correcto != 1)
            {
                  printf("\n Error en la escritura");
                  correcto = 0;
            }
}
```

La contraparte de `fwrite()` es `fread()`. Su función prototipo es la siguiente:

*size_t fread(void *buffer, size_t número_de_bytes, size_t cuenta, FILE *puntero_archivo)*

Donde:

size_t Es un tipo definido en stdio.h y es semejante al tipo unsigned

buffer Es un puntero a la localidad de memoria a donde serán escritos los datos procedentes del archivo.

numero_de_bytes Número de bytes que se leerán del archivo

cuenta Es el número de elementos con *numero_de_bytes* de longitud que se leerán del archivo

puntero_archivo Corresponde al archivo del cual se leerá la información.

Para leer los datos escritos en el ejemplo precedente se escribiría el siguiente código:

```
#define NUMALUMNOS  25 /* Esta macro indica el numero de alumnos en el grupo*/
FILE *arch;
register int contador;
int correcto = 1;    /* bandera que indica si existió algun error en la escritura
                        al archivo */
```

```
/* Suponemos que en algun lugar se abre el archivo y almacena su puntero en ARCH*/

for (contador = 0 ; contador < NUMALUMNOS && correcto; contador++)
{
      correcto = fread(&alumno[contador],sizeof( struct inf_alumno),1,arch)
            if (correcto != 1)
            {
                  printf("\n Error en la lectura");
                  correcto = 0;
            }
}
```

Los archivos que hemos manejado hasta este momento han accesado la información en forma *secuencial*, es decir, un registro tras otro en una estricta secuencia. Si deseamos accesar el décimo registro primero debemos realizar la lectura de los nueve registros precedentes. En lenguaje C el acceso a una registro directo se logra a través de la función *fseek()*, la cual tiene la siguiente función prototipo:

*int fseek (FILE , *puntero_archivo, long numero_bytes, int origen)*

Donde:

puntero_archivo	Corresponde al archivo en el cual se realizará una operación de lectura/escritura
numero_bytes	Es la cantidad de bytes que se movera el puntero del archivo con respecto a la dirección de inicio de archivo.
origen	Indica el origen del desplazamiento del puntero que se puede realizar a partir del principio del archivo, de la posición actual o de el fin del archivo

La especificación del origen se puede realizar a través de las siguientes macros definidas en stdio.h

SEEK_SET	Principio del archivo
SEEK_CUR	Posición actual
SEEK_END	Fin del archivo

La función fseek() retoma cero si se concluyó la tarea sin errores.

En general esta función es especialmente útil cuando todos los registros poseen la misma longitud, ya que para accesar el n-ésimo registro bastará con multiplicar la dimensión en bytes de cada registro y despues posicionarse con la función *fseek()*.

Ejemplo:
Para leer un registro cualquiera del archivo que almacena la estructura inf_alumno definida en párrafos anteriores bastará con escribir el siguiente código:

```
long num_alumnos, posicion;
FILE *arch;
.
.
.
/* Suponemos abierto el archivo */
printf("\n Digite el número de registro del alumno");
scanf("%ld",num_alumno);
```

```
num_regis= (num_alumno-1L)*sizeof(inf_alumno);
fseek(arch, numregis, SEEK_SET);

/* Operacion de lectura o escritura */
.
.
.
```

II.5.4 Archivos de datos formateados

Las funciones anteriores realizan una escritura o una lectura con formatos que son totalmente libres del programador y que son definidos por el compilador. Cuando el programador desea grabar la información en un formato predefinido por él mismo, se puede valer de las funciones **`fprintf()`** y **`fscanf()`**.

Básicamente **`fprintf()`** y **`fscanf()`** funcionan de la misma manera que lo hace **`printf()`** y **`scanf()`** respectivamente. Sus funciones prototipo son:

*int fprintf(FILE *puntero_archivo, const char *cadena_de_control, lista_de_variables);*

*int fscanf(FILE *puntero_archivo, const char *cadena_de_control, lista_de_variables);*

II.6 Programación de bajo nivel

Una poderosa herramienta de trabajo en el lenguaje C es el acceso a los bits que conforman una palabra cualquiera. Gracias a esta característica es posible precindir en una gran cantidad de programas de la escritura de versiones en lenguaje ensamblador. El lenguaje ensamblador, hace a un programa totalmente dependiente de una máquina, y por tanto poco portátil. Programar dichas funciones en lenguaje C asegura la portabilidad de los programas que se escriban.

Generalmente se escribe en lenguaje ensamblador para escribir procesos que sean más veloces. Los optimizadores de código y de velocidad que acompañan a los compiladores de C, hacen difícil pensar en las mismas rutinas escritas en ensamblador con una reducción en el tiempo lo suficientemente significativa como para justificar el esfuerzo invertido.

II.6.1 Operaciones de bits

Los operadores que se pueden utilizar para manipular bits son los siguientes:

Operador	Función
&	Y lógico a nivel de bits
|	O lógico a nivel de bits
^	O exclusivo a nivel de bits
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda
~	Complemento a 1

Estos operadores se aplican directamente sobre los bits que conforman un byte para los tipos char e int. No es posible aplicar estos operadores a dostos mas complejos como lo puede ser float, double, struct, etc.

OPERADORES & , | y ^

Estos operadores tiene la misma tabla de verdad que los que se han estudiado en los apartados precedentes. La única diferencia es que ahora se aplica bit por bit. Por ejemplo. Si deseamos colocar a cero los dos bits menos significativos de un byte, pero conservar el valor de todos los demas, podemos escribir una asignación como la siguiente:

```
/* ej61 */
main()
{
int resultado,byte;

byte=0x6;
resultado=byte & 0xFFFC;
printf("%x %x",byte,resultado);
}
```

En este ejemplo se asigna a la variable byte el valor de 6 (0110 binario). El numero 0xFFFC está representado en hexadecimal. Sabemos que el último digito (C) se representa como 1100 en forma binaria. De tal forma que al multiplicarlo por el valor de resultado nos entregará los dos bits menos significativos con cero y el resto de los bits de la variable byte con el valor que conservaban. Por simplicidad solo representaremos los cuatro bits menos significativos:

```
byte      0110
0xC       1100
resultado 0100
```

El resultado desplegado en pantalla sera por tanto

```
6 4
```

Si en el mismo ejemplo utilizaramos el operador | en lugar del operador &. El resultado sería el siguiente

```
byte      0110
0xC       1100
resultado 1110
```

Este operador coloca a uno los bits que en el número tienen un número uno y conserva el valor de los bits que se encuentran en los ceros: El resultado desplegado en pantalla será:

```
7 fffe
```

La tabla de verdad del operador ^ dice que dos bits distintos arrojarán un uno como resultado y dos bits iguales arrojarán un cero.

El operador ^ en el programa anterior arrojaría los siguientes resultados:

```
byte        0110
0xC         1100
resultado   1010
```

```
7 fffa
```

OPERADORES DE DESPLAZAMIENTO <<, >>

Un desplazamiento de bits implica que todos los bits se muevan hacia un lado. El operador >> moverá todos los bits a la derecha, en tanto que el operador << los desplazará a la izquierda. Cuando un byte es desplazado los bits nuevos están colocados a cero. La forma general para utilizar el desplazamiento es:

variable operador_de_deplazamiento numero de bits a desplazar

Ejemplos:
Supongamos una variable llamada `byte` con la siguiente informacion:

11100111

El resultado de las siguientes operaciones se muestra a continuación:

```
byte >> 1       01110011
byte << 1       11001110
byte >> 3       00011100
byte << 4       01110000
```

En general puede observarse que un desplazamiento a la derecha divide la cantidad entre dos, en tanto que un desplazamiento a la izquierda multiplica la cantidad por dos.

EL OPERADOR ~

El operador ~ es unario y obtiene el complemento a uno de la variable, es decir, este operador coloca todos los unos a cero y todos los ceros a uno.

```
  byte      11100111
 ~byte      00011000
```

Ejemplo:

En diversos equipos de entrada, una tecla puede tener asociados una secuencia de caracteres que la representan. En las computadoras personales, cuando se lee una tecla con una variable entera la parte alta de la palabra corresponde al código de inspección de la secuencia que nos interese. (cntrl, alt, etc) La parte baja de la varable corresponde a la letra en sí. El siguiente programa devuelve exclusivamente la parte baja del caractér y elimina el código de inspección.

```
int getkey(void)
/* Usa el BIOS para leer el proximo caracter del teclado */
{
 int key, lo, hi;

 key = bioskey(0);
 lo = key & 0X00FF;
 hi = (key & 0XFF00) >> 8;
 if (lo == 0 )
      lo = hi + 256;
 return(lo);
} /* getkey */
```

II.6.2 Campos de bits

Cuando se operan interrupciones a la familia de microprocesadores 80x86 o cuando se realizan aplicaciones que tienen que ver con caracteristicas del hardware, como en una transmisión de datos por uno de los puertos, es frecuente tener que acceder a los bits que conforman un byte.

La forma general para definir una estructura cuyos miembros accedan directamente a los bits es la siguiente:

struct etiqueta
{
tipo identificador: longitud ;
tipo identificador: longitud;
tipo identificador: longitud;
.
.
.
}lista_de_variables;

Donde:

tipo	Puede ser int, unsigned o signed.
longitud	Es el número de bits que conformarán el campo.

Los campos de un solo bits deben declarase como unsigned ya que un bit no puede tener signo.

Programa:
Utilizando la interrupcion 11h de DOS escriba un programa que liste el equipo conectado a un computador personal que trabaje bajo DOS.

```
/* UNAM - ENEP
Detector de equipo instalado
13/marzo/1993 */
#include <stdio.h>
#include <conio.h>
#include <dos.h>

#define EQUIPO 0x11
```

```
int main(void)
  {

      struct BITS
     {
      unsigned discos: 1;
      unsigned       : 1;
      unsigned ram   : 2;
      unsigned video : 2;
      unsigned n_disc: 2;
      unsigned dma   : 1;
      unsigned rs2_32: 3;
      unsigned juegos: 1;
      unsigned n_impr: 2;
     };

     union
     {
      struct WORDREGS x;
      struct BYTEREGS h;
      struct BITS     b;
     }regbits;


     clrscr();
     printf("\n\n Rastreador de sistema\n\n\t Lista de equipo");
     printf("\n\t==============\n\n");



     int86(EQUIPO, &regbits, &regbits);
     printf("\n\t Unidades de disco flexible       :");
     if (regbits.b.discos)
     {
      printf(" presentes");
     }
     else
     {
      printf(" ausentes");
     }
     printf("\n\t Numero de unidades de disco duro : %u",regbits.b.n_disc);
     printf("\n\t Numero de puertos serie          : %u",regbits.b.rs2_32);
     printf("\n\t Numero de impresoras             : %u",regbits.b.n_impr);


     printf("\n\n\t Modo de video      : ");
     switch(regbits.b.video)
     {
      case 0x1:
            printf(" 40 x 25 color");
            break;
      case 0x2:
            printf(" 80 x 25 color");
            break;
```

```
    case 0x3:
          printf(" Monocromático");
    default:
          printf(" Desconocido");
}


if (regbits.b.dma)
 printf("\n\t Chip DMA            : ausente");
else
 printf("\n\t Chip DMA            : instalado");


printf( "\n\n\t Adaptador de juegos:");
if (regbits.b.juegos)
 printf(" instalado");
else
 printf(" ausente");
```

CAPITULO

III

METODOLOGIA DE LA PROGRAMACION ORIENTADA A OBJETOS

Consideremos un mapa como un modelo de un territorio. Para que nos sea útil, el mapa debe ser más simple que el territorio que modela. Si incluyera cada detalle, este podría ser del mismo tamaño que el territorio y fallaría en su propósito.

Rebecca Wirfs-Brock

III.1 La programación orientada a objetos

Aunque la programación orientada a objetos no es realmente nueva, si ha tomado un gran auge en los últimos años. Representa en si misma un enfoque que reune en una sola filosofía los mejores atributos de las escuelas que le han precedido históricamente.

En este capítulo abordaremos los conceptos que la fundamentan y complementaremos el estudio con una introducción a C++. Este capítulo no pretende mostrar en forma exahustiva este lenguaje debido al espacio necesario para abordar un tema tan extenso, en su lugar se estudiarán los conceptos básicos del lengueje para mostrar los conceptos de la programación orientada a objetos conforme estos se vayan exponiendo.

III.1.1 Definición del problema

COMPLEJIDAD DEL MUNDO REAL

El mundo real es un conjunto de elementos interrelacionados entre sí. Philip J. Farmer comenta que *"todos los acontecimientos, y por tanto, todos los hombres, están interconectados en una tela de araña irrompible. Lo que un hombre hace, no importa cuán insignificante parezca, vibra por las hebras y afecta a todos los demás hombres"*. Cada acontecimiento lleva implícita una complejidad inherente a esta interrelación.

La realización de tareas que para los seres humanos son poco menos que triviales, se convierten en tareas muy complejas cuando se decide implantarlas en un ordenador, Reconocimiento de voz e imágenes, comprensión de instrucciones habladas, traducción de un lenguaje a otro... Y esto hablando solo de tareas altamente especializadas, no estamos incluyendo el desarrollo de los cerebros positrónicos, ni la implantación de las tres leyes de la robótica que nos plantea Asimov en sus escritos y que incluirían varias de estas tareas reunidas en un sistema integral, general y flexible.

La complejidad del los procesos que existen en el mundo real se trasluce cuando un analista de sistemas pide a su cliente que explique lo que desea o que cuente en breves palabras en que consiste el trabajo que se pretende realizar con la computadora. Generalmente el usuario del sistema que será implantado tiene un mar de ideas que no reflejan las especificaciones que el analista necesita para elaborar la lista de requerimientos.

Un sistema real es la unión de muchas interrelaciones que se abocan a uno o diversos fines con múltiples resultados. El usuario suele ver su sistema como *"el inventario de almacen"* o *"la cuenta bancaria"*. Su nivel de abstracción es tan alto que difícilmente comprende la tremenda complejidad que entraña una actividad tan *sencilla* como pedir dinero al cajero automático en la ciudad de Monterrey aunque su cuenta se encuentre en la ciudad de México.

LA CRISIS DEL SOFTWARE

En el apartado I.1.3 se comentó que los sistemas comenzaron a crecer en tamaño y en complejidad. Fué necesario investigar sobre nuevas formas de realizar software. En la decada de los 70's la programación estructurada tuvo su gran auge y marcó el camino para que se desarrollara el enfoque del Análisis estructurado.

Las aplicaciones han continuado desarrollandose desde entonces. Con el establecimiento de las diversas metodologías para la creación de los productos de programación, más y mejores programas se encuentraron en el mercado para satisfacer las necesidades de *las empresas*.

La nueva revolución de los computadores la marco el microprocesador Z-80 que permitió el diseño de computadores caseros. Para cuando INTEL lanzó al mercado su microprocesador 8088, IBM guardaba celosamente el diseño de la computadora que se conoce actualmente como PC (personal computer). El computador personal permitió que la computadora pasara de las oficinas de las grandes empresas a los hogares de millones y millones de personas en todo el mundo. Tan solo en 1990 la venta de computadoras personales en todo el mundo alcanzó una cifra cercana a los 80 miles de millones de dolares.

Nos comenta Robert Lucky:

En el futuro se van a conseguir computadores como premios en los paquetes de cereales para el desayuno. Usted los tirará a la basura, porque su casa estará invadida de ellos.

Hoy dia podemos ver que las computadoras se encuentran ya en en cada lugar que requiere de un procesamiento de información. Cada uno de esos computadores exige de software que la haga productiva. La posibilidad de que cada empresa posea una computadora hace necesaria en cada empresa una persona que se haga cargo del desarrollo de los sistemas *a la medida* de la empresa. Cada empresa tiene diversas necesidades definidos por sus propias políticas, los paquetes de software han solventado en parte algunas de esas necesidades, pero en general nunca llegan a satisfacer plenamente al cliente.

Entendiendo por *profesionales* a aquellas personas que se dedican a una actividad, la demanda de los mismos se ha incrementado en forma considerable en los últimos años. El nivel de capacitación que las empresas requieren de los *profesionales* ha decaido considerablemente. La oferta y la demanda no están caminando de la mano. Por otra parte las grandes empresas de software desarrollan software sistemas cada vez más sencillos de manejar. El nuevo mercado de los sistemas de cómputo se muestra hambriento de programas cada vez más poderosos y más *sencillos de manejar.*

Un programa sencillo de manejar requiere que este se comporte lo más inteligentemente posible y que evite al usuario tomar decisiones. Los medios de información masivos han hecho creer a las personas que las computadoras lo pueden todo y que solo es necesario oprimir una tecla para ello. Esta desinformación de los usuarios acarrea conductas intransigentes hacia el experto en sistemas. En más de una ocasión he escuchado decir a los usuarios: *"¿No habría forma de decirle nada más que lo haga y ya?"* . O expresado de otra forma: *"¿No habría forma de que pensara por mí?"*

Los sistemas son entidades muy complejas en su naturaleza misma. Sin embargo es labor del especialista, crear la sensación de sencillez. Esta es la tendencia actual en cualquier rama de la tecnología. Todo se ve reducido a sencillas cajas negras.

En los capítulos precedentes se ha puesto de manifiesto que el software posee una gran capacidad para modelar una gran variedad de situaciones. Cualquier proceso computable es factible de ser modelado, pero el costo de ese modelado es una función directa de los límites que se impongan al sistema. *Entre más inteligente parezca un sistema, más complejo será el código y más tiempo se invierte en su escritura, su desarrollo y su posterior mantenimiento.*

La creciente demanda de software y el imperativo aumento de su complejidad exige de los desarrolladores la creación de software nuevo en muy cortos periodos de tiempo. La reutilización del software se vuelve imperiosa. El problema estriba en que los desarrolladores de software suelen pensar en función de la aplicación a la que desean dar solución y no en la posibilidad de hacer una rutina que puedan utilizar cada vez que requieran iniciar un nuevo sistema.

Los productos comerciales de software suelen especificar en la garantía que el producto cambiará sin previo aviso. Los usuarios exigen de los productos más y más características. Tomese el caso de los procesadores de texto de la PC. Iniciaron con edlin que es aún un procesador de texto en línea muy

rudimentario pero que servía para su propósito. Los usuarios exigieron de un software que les permitiera moverse al través de toda la pantalla, este concepto introdujo los procesadores de texto a pantalla. El siguiente paso fué la implantación de las características de búsqueda, corrección de ortografía, diccionario de sinónimos, diversos tipo de letra, etc . Finalmente nos encontramos ante los procesadores de visualización de fuente verdadera (truetype) en el cual podemos insertar gráficos, y ver el documento tal y como se imprimirá en la hoja. El software no es de ninguna manera estático. Como se puede apreciar, la gran demanda de sotfware hace que sean necesarios métodos para reutilizar el software y lo hagan lo más flexible que sea posible. No es posible empezar de cero cada vez que se desee crear una nueva aplicación.

En algún momento de nuestra vida profesional nos encontramos ante una interrogante que se tiene que resolver bajo presión: *¿Que se tiene que hacer cuando se tiene un programa que falla en una terminal que da servicio, y ese terminal tiene una cola de 20 personas?* .Muchos programadores tienen por respuesta : *Poner un parche, un salto incondicional, crear una variable global o cualquier cosa que solucione el problema.* Bajo esta filosofía no es nada extraño que los sistemas tengan un tiempo de vida media de cinco años. Solo el tiempo necesario para que, *de parche en parche*, el sistema se destruya a sí mismo agobiado por los múltiples conflictos que las variables generarán o por volverse totalmente incomprensible, aún para la persona que le da mantenimiento.

Esta es una situación que se ha venido presentando cada vez con mayor frecuencia en las empresas. Por una parte el software requerido es cada vez más complejo, la calidad de los profesionales ha decaído y finalmente el tiempo empleado para el diseño es considerado como una pérdida de tiempo que redunda en software de mala calidad con altos costos de mantenimiento y corta vida productiva. Esta es la llamada *crisis del software.*

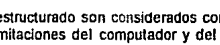
El enfoque estructurado no ha dado solución a muchos de los problemas a los que se enfrentan actualmente los desarrolladores. Una parte de esto es debido al desconocimiento de lo que realmente es la programación estructurada, a los enfoques incorrectos, o la negligencia en la implantación de los módulos. *La programación estructurada nació para evitar los problemas, no para resolverlos.* Si la programación estructurada es aplicada sin un conocimiento profundo de lo que realmente es en escencia, lo más probables es que no produzca ningún beneficio real. Por otra parte, el enfoque estrictamente algorítmico del desarrollo estructurado (y ahora si estamos incluyendo al análisis estructurado), priva al analista de muchas herramientas que le permiten diseñar sistemas orientados al *problema* y no a la solución.

La programación y el analisis estructurado son considerados como enfoque orientados a la solución porque se realizan pensando en las limitaciones del computador y del software que será implantado en el computador. Lo más importante para la programación estructurada es el programa, lo más importante para el análisis estructurado son los procesos que modican la información.

PARADIGMAS DE COMPUTACIÓN

. Paradigma (Latín *paradígma*, Griego *paradeigma)* es una palabra que significó en su forma original, *ejemplo ilustrativo.* Thomas Kuhn expandió la definición a: *conjunto de teorías, estándares y métodos que en conjunto representan una manera de concebir el conocimiento.*

La programación estructurada y la programación orientada a objetos son distintos paradigmas porque representan formas distintas de ver un mismo conocimiento. Sus enfoques cambian en cuanto a la forma como se abstraen los datos que se representarán en forma de un programa. Se habia mencionado que un programa puede ser visto básicamente como dos entidades distintas:



La programación estructurada abstrae las formas de control más comunes de los programas. Este

paradigma está orientado a los procedimientos y se implanta pensando en lenguajes imperativos. La programación estructurada esta enfocada a los programas, es decir, al *cómo* se solucionará un problema. No contempla directamente el manejo que recibirán los datos . Los diagramas de Wamier-Orr en cambio si son eminentemente orientados a los datos.

La programación orientada a los objetos no se preocupa por el cómo se implantará un módulo. La primer pregunta que se realiza en el enfoque de objetos es:*¿Sobre quién* se efectuará un procedimiento?. La programación orientada a objetos (POO) no es en forma estricta un punto de vista totalmente orientado a los datos. Esto se verá justificado con amplitud cuando definamos lo que es un objeto.

La programación orientada a objetos representa una forma totalmente distinta de ver la computación. Podemos decir que sus principios están cimentados en la programación y el análisis estructurado, pero hacen una serie de refinamientos e implantan una serie de características que la colocan como una paradigma totalmente nuevo.

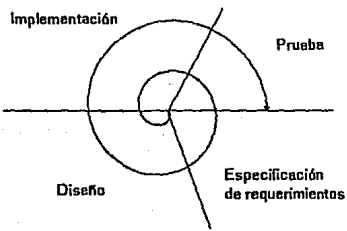
Estos dos enfoques no son lo únicos existentes, pero los programadores e inclusive los analistas se encuentran encerrados en el paradigma con el cual fué creado el lenguaje con el que normalmente trabajan (y generalmente se trabaja un solo lenguaje). Bobrow y Stefik listan cinco tipos de abstracción:

- Orientado a los procedimientos	Algoritmos
- Orientados a los objetos	Clases y objetos
- Orientados a la lógica	Metas, espresados en calculo de predicados
- Orientados a reglas	Reglas SI - ENTONCES
- Orentados a constrefilmiento	Relaciones invariantes

Cada uno de estos modos de entender el modelado por programas proporcionan una visión distinta del mismo conocimiento. No se puede establecer que alguno de estos enfoque sea superior al otro ya que cada uno sirve para la resolución de distintos problemas. Queda totalmente fuera de las posibilidades de estos apuntes hacer un estudio de cada uno de estos paradigmas.

CICLOS DE CREACION DEL SOFTWARE

Rebecca Wrift-Brook sugiere que la metodología de creación de software que se utiliza en forma tradicional se puede visualizar como una espiral que luciría de la siguiente manera:

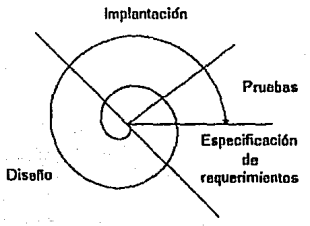


Ciclo de vida tradicional del software

La espiral muestra que una gran parte del tiempo se invierte en las pruebas de la implantación y la prueba del sistema...cuando se invierte tiempo en el diseño.

Cuando se está pensando en implantar un sistema orientado a objetos es muy difícil omitir la parte de diseño ya que es necesario planear cuidadosamente las características que cada parte del sistema deberá de cumplir. La POO está claramente enfocada a la *reutilización* del código. Su gran meta es implantar lo mejor posible la complejidad del mundo real, abatiendo costos de mantenimiento y desarrollo.

La espiral correspondiente a la POO es la siguiente:



Ciclo de vida del software orientado a objetos

Tal y como se puede observar, la etapa de diseño es mucho mayor que en el enfoque tradicional. Esto trae como consecuencia un planeamiento más cuidadoso de los componentes del sistema. Se diseña pensando en su posterior reutilización.

La POO no es la solución a todos los problemas. El mero hecho de que un programa sea escrito en un lenguaje orientado a objetos no garantiza que el programa sea más barato, ni mas flexible.

Ahora bien, algunas compañías han optado por reescribir sus programas en un desendiente directo de un lenguaje imperativo. Los que escribían en lenguaje C, ahora escriben en C++ por mencionar un caso. Piensan que la transición entre un enfoque imperativo y el enfoque a objetos se dará de una manera más suave. Nada tan lejos de la realidad como esto. La POO exige del programador un nuevo punto de vista y una verdadera disciplina mental para escribir programas.

En cierta ocasión un colega con quien comentaba el material de estos apuntes me preguntó con mucho interés: *Y cuando llegues al capítulo de programación orientada a objetos ¿Cómo le vas a hacer para decir que todo lo del capítulo uno ya no sirve?* Muchos libros, y otros tantos profesionales consideran a la programación estructurada ha quedado totalmente en desuso, que la programación orientada a objetos desacredita totalmente a la programación estructurada. Estas opiniones sin embargo tienen mucho de fanatismo y poco espíritu crítico.

El análisis estructurado es totalmente ortogonal al diseño orientado a objetos, pero no se niegan a sí mismos. Son simplemente formas *distintas* (no se usó la palabra contradictorias) de ver el diseño e implantación de un programa. Se puede lograr un diseño híbrido utilizando los lineamientos expresados por Constantine en *Objetos, funciones y extensibilidad de los programas.* o el enfoque utilizado en estos apuntes.

Al respecto Grady Booch se cuestiona:

> *¿Cuál es la manera correcta de descomponer un sistema complejo? ¿Por algoritmos o por objetos? Actualmente, esta es una cuestión por resolver, porque la respuesta correcta es que ambos puntos de vista son importantes; el punto de vista algorítmico enfatiza el ordenamiento de eventos, y el punto de vista de la programación orientada a objetos realza los agentes que de una u otra forma serán causa de acción o son los sujetos con los cuales esas operaciones actuarán.*

OBJETOS Y RESPONSABILIDADES

El enfoque orientado a objetos puede obtenerse como una consecuencia de los conceptos que se expusieron en el capítulo uno de estos apuntes. Recapitulemos los conceptos más importantes:

Modulo: Hemos definido a un módulo como una *función* que posee un conjunto finito de entradas y un conjunto finito de salidas que realiza una tarea *específica.*

Principio del diseño estructurado: Todos los elementos que tengan relación entre sí deberán estar contenidos en un solo módulo. Los elementos que no tengan relación con la función del módulo, no deberán estar incluidos en dicha función.

Cohesión: Las fuerzas que unen a los elementos de un módulo deben ser lo más fuertes que sea posible. Para lograr esto todos los elementos deberán estar relacionados para realizar un solo objetivo.

Acoplamiento: Cada módulo debe unirse a los demás con la menor fuerza posible y debe comunicarse con los otros módulos solo a través de aquellos parámetros que realmente son necesarios. El resto de la información debe ser manejada con variables locales.

Teorema de la estructura: Un programa puede ser construido con tres estructuras de control.

Programa propio: Un programa debe tener un solo punto de entrada por arriba, leerse de arriba a abajo y tener un solo punto de salida por abajo.

Estos conceptos están claramente orientados al algoritmo con el cual se resolverá un problema. Antes de codificar la primer línea de cualquier programa es necesario establecer cuales son las funciones que deberán cumplir los módulos, establecer correctamente las interfaces de cada uno de ellos, y vigilar de cerca que la cohesión de los módulos sea lo más fuerte posible.

La programación orientada a objetos da un paso más allá al extender el principio del diseño modular. La pregunta que se hace la POO es: ¿Acaso los datos de un programa, no forman también parte del programa? ¿Acaso un módulo no posee una cantidad de información (que no tiene que ver con las estructuras de control) que le permiten realizar sus tareas?

La respuesta inequívoca es sí. Un módulo que maneje la impresora *conoce* cuales son los códigos que deberá enviar para cambiar el ancho de la letra, para imprimir en negritas o cambiar el tipo de letra. Un módulo que elabore un reporte puede llamar a la función que maneja la impresora sin tener que enviar como parámetros todos estos códigos. No es *responsabilidad* del módulo de reportes conocer dichos códigos.

Dentro de la POO cada objeto posee un conocimiento y es responsable de determinadas acciones. Los enfoques de diseño orientado a objetos más recientes hacen mucho énfasis en la arquitectura cliente-servidor. Cada objeto puede pedir servicio a otro objeto y este será responsable de entregar a su cliente una solución satisfactoria a la demanda.

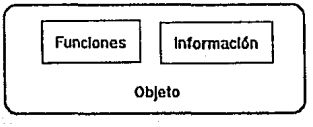
Un módulo es un programa que realiza una función específica. Es decir, un módulo posee un *método* de solución para una tarea que se le ha sido encomendada. Un módulo no solo posee los *métodos* para realizar su trabajo, también posee el conocimiento suficiente para realizar ese trabajo. Por otra parte el módulo que maneja la impresora, puede encontrarse en una serie de estados distintos: esperando que se coloque en línea la impresora, esperando a que se encuentre lista para recibir, o puede encontrarse enviando datos. Cada una de estas situaciones representa un *estado* distinto.

El elemento primitivo de la programación orientada a objetos es un objeto. ¿Qué es entonces un objeto?. A reserva de definirlo en forma más amplia con posterioridad podemos decir, sin quedar faltos de presición, que *un objeto es una entidad que tiene una función específica de la cual es responsable , y la información necesaria para cumplir con esa tarea. Un objeto posee estados, conducta e identidad.*

Resulta curioso hablar de conducta en un programa, pero cuando se le analiza con más calma se verá que los conceptos no están errados. En general se habla de una conducta cuando una entidad reacciona ante un estímulo de un determinada manera. Un objeto responderá ante el mundo exterior comportandose de una determinada forma que será función del estado del objeto y del mensaje que reciba de ese mundo exterior. En el lenguaje de la POO la función de un módulo se conoce como *conducta* del módulo. La conducta de un objeto será por tanto el conjunto de funciones que conformen a dicho objeto

Aunque dos gemelos sean idénticos, cada uno posee un caudal de conocimientos que lo diferencia del otro. Cada gemelo posee su propia idéntidad. De la misma forma, cada objetos es distinto de todos los demás objetos existentes en un momento dado. El estado de sus variables, los mensajes recibidos y la manera como reaccionará ante un mensaje, le dan su propia idéntidad.

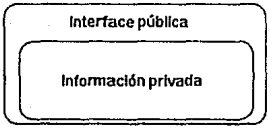
Cada objeto dentro de un sistema debe tener una responsabilidad bien determinada. Su conducta estará basada en las especificaciones que determinan su responsabilidad. El objeto deberá tener toda la información necesaria para cumplir con dichas responsabilidades. Agrupar a la funciones y la información en una entidad conocida como objeto es *encapsular* la identidad del objeto dentro de límites bien establecidos. Estos límites le dan la identidad necesaria para ser distinto de cualquier otro objeto.



El encapsulamiento de la información y las funciones son la primera diferencia entre la programación tradicional y la POO. El encapsulamiento nos permite tratar al objeto como una entidad independiente y única de más sencilla manipulación. Es una caja negra con una estructura distinta a la de los módulos. Observese que no hemos negado ninguno de los principios expuestos hasta este momento, los hemos ampliado.

Cuando una persona acude con un médico por causa de una enfermedad, uno espera del médico una serie de conductas apropiadas en él. Esperamos que nos haga un buen chequeo y que en base a eso diagnostique la enfermendad. El médico no consulta a la enfermera, ni al la persona que limpia su consultorio para realizar el diagnóstico. Su identidad propia lo hacen tener todo el conocimiento requerido para cumplir con su labor. Cuando el diagnóstico cae en funciones que están fuera de su competencia (responsabilidad) como lo es un análisis de sangre, entonces se envía un mensaje a una entidad que sí puede cumplir dicha responsabilidad: un laboratorio clínico. El médico solo envía la información necesaria al laboratorio y este a su vez responde solo con la información necesaria para que el médico pueda cumplir con su responsabilidad.

Un objeto posee una forma interna de comunicación y una externa. Es decir, un objeto posee una interface pública con la cual se comunica a través de *mensajes* con los otros módulos. Por otra parte el objeto posee una información privada que oculta al resto de los objetos y que solo a él es de utilidad para cumplir con su responsabilidad. Una característica muy importante de los objetos es el *ocultamiento de la información.* Un objeto que no poseea los métodos suficientes para cumplir con una función, puede comunicarse con objetos que lo ayudan a cumplir dicha labor. Un objeto puede entonces, tener *colaboradores* .



Ocultamiento de la información

En la programación tradicional es deseable que las variables que no necesiten ser globales se declaren siempre como variables locales, es decir, como información que solo el módulo que las ha creado puede conocer. También es deseable que un módulo se comunique con los demás solo con los datos estrictamente necesarios. Esto también existe en la POO. En cualquier caso con esto se logra una mayor flexibilidad en los componentes del sistema. En la programación tradicional un módulo debía ser accesado a través de una lista de parámetros, en la POO se accesa a través de un mensaje, que es una forma alterna de enviar parámetros.

Tal y como puede observarse, el encapsulamiento es un refinamiento del principio del diseño modular. Los conceptos vertidos sobre acoplamiento y cohesión tienen su máxima vigencia cuando se trata a los objetos ya que estos solo se pueden comunicar entre sí por medio de mensajes. Cada servidor ocultará a su cliente la manera como realiza su trabajo mostrando tan solo una interface pública que debe ser cuidadosamente diseñada.

ANÁLISIS ORIENTADO A OBJETOS

El capítulo uno hizo énfasis en que cada módulo debía cumplir una función bien delimitada. En el libro *Designing Objet-Oriented Software* ,Rebeca Wirfs-Brock et. al. propone un método para realizar un buen diseño orientado a objetos. Este consiste en la asignación de responsabilidades a cada uno de los objetos que conformarán el sistema.

Una responsabilidad incluye dos puntos clave:
- El conocimiento que el objeto mantiene, y
- Las acciones que un objeto puede realizar.

Las responsabilidades se pueden obtener de la lista que requerimientos del sistema. En dicha lista se especifica claramente cuales son los sujetos sobre los cuales se actuará y que provocarán las acciones que los objetos realizarán cuando reciban un mesaje. La definición del problema o análisis de requerimientos puede obtenerse con cualquiera de las técnicas examinadas hasta el momento.

III.1.2 Identificación de objetos y clases

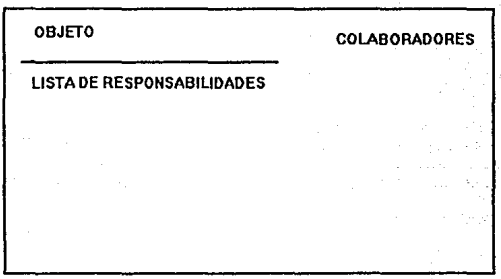
IDENTIFICACIÓN DE LOS OBJETOS

La identificación de los objetos se obtiene directamente de la lista de requerimientos. Si se posee es necesario leerla muy bien hasta entender claramente que es lo que se desea hacer y sobre quien se debe actuar. Si la lista no se posee es necesario realizar un enunciado que liste en forma exahustiva los requisitos que el software debrá cumplir.

Supongamos que deseamos implantar una terminal de punto de venta. La lista de requerimientos podría ser la siguiente:

Cuando la cajera pulsa el boton de inicio, se debe imprimir el encabezado del comprobante de compra (tiket), El terminal entonces estará listo para acepta la lista de los productos de la tienda. En cualquier momento lla cajera podrá pedir un subtotal. Cuando se indique el fin de captura de mercancías se debe obtener un total, recibir la cantidad con que se paga, abrir la caja del cambio y depiegar el cambio que se devolverá al cliente.

En tarjetas de 10 x 15 centímetros anotamos cada una de las responsabilidades de las funciones que encontremos:



Tarjeta de responsabilidades

En primer lugar encontramos que necesitamos colocar a cero los totales y subtotales así como la impresión del encabezado.

IniciaCompra	**IniciaImpresora()**
Colocar a cero los contadores	
Imprimir encabezado el el recibo	

Posteriormente es necesario un objeto que capture cada uno de los productos

CapturaProductos	**ImprimeProducto()**
	ImprimeSubtotal()
Acepta la cantidad que digite la cajera	
Aplica descuento al producto	
Actualiza totales y subtotales	
Muestra subtotal	
Acepta una cancelación	

Finalmente tenemos un objeto que obtendrá los totales y aplicará los descuentos e impuestos globales

ObtieneTotales	**ImprimeTotal()**
	DesglozaCambio()
Imprime cantidad total neta	
Aplicar descuentos totales	
Aplica impuestos	
Calcula cantidad total bruta	

Cada una de estas tarjetas será refinada conforme el proceso de diseño avance. Es necesario averiguar que conocimiento requiere cada objeto para cumplir con sus responsabilidades.

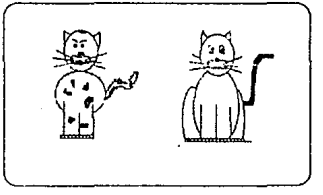
CLASES

- ¡Que bonito está tu gato!- exclamó la niña de cinco años a su primo Juancito de cuatro.
- ¡No seas tonta! - exclama el niño furioso - Ese no es un gato, es Manchitas.

Esta anécdota puede arrancarnos una sonrisa en primera instancia. Pero pensemos un momento en el siguiente caso. La programación tradicional posee un operador de suma para los datos básicos, pero necesita que un módulo se llame `suma_matrices()`, y otro `suma_vectores()` y un tercero `suma_complejos()`. Cada función es distinta de todas los demás aunque nuestra capacidad de abstracción nos grita que se trata de lo mismo.

Se ha insistido mucho en el papel que juega la abstracción dentro del diseño de sotware. En la POO la abstracción es la herramienta clave de diseño porque los seres humanos solucionamos nuestros problemas cotidianos a través de la abstracción. En la anécdota expuesta al inicio de este apartado, la niña había obtenido las características escenciales de todos los gatos que había visto con anterioridad y los había agrupado en una categoría aparte. Probablemente la niña jamas había visto a Manchitas, pero lo identificó claramente como un miembro de esa categoría.

Uno de los procesos más comunes que los seres humanos empleamos para solucionar problemas es la clasificación de los objetos dentro de categorías más amplias que los abarquen. Estas categorías de objetos reciben el nombre de *clases* . Una clase agrupa a todos aquellos objetos que poseen características similares. Un objeto que pertenece a una clase recibe el nombre de *instancia* . *Manchitas* es una instancia de la clase llamada *gatos*. Las funciones `suma_matrices()`, `suma_vectores`, `suma_complejos()` y el operador + son instancias de la clase *suma*.



Objetos con características similares forman una clase

La clasificación de los objetos dentro de clases se obtiene relacionando aquellas características comunes a los objetos. Cada clase debe poseer un conjunto de conocimientos que le permitan realizar las responsabilidades que le sean conferidas. Por ejemplo, la clase *sumas* que hemos definido, debe tener todos los conocimientos necesarios para implantar una adición entre dos operandos, sin importar el tipo de argumento con el cual la sea llamada la función. Cuando se envía un mensaje a una clase, esta determina cual de sus miembros tiene la capacidad necesaria para resolver la tarea que le ha sido encomendada.

Las tarjetas que presentamos anteriormente se conocen como tarjetas CRC. Una tarjeta CRC recibe ese nombre porque en ella se registran las Clases, las Responsabilidades y los Colaboradores. Cuando se han obtenido los objetos básicos del sistema es necesario agruparlos en clases.

Podemos observar que los colaboradores que hemos definido en el ejemplo de la caja punto de venta tiene una característica común: todos están relacionados con la impresión de resultados. Es evidente que todos ellos requerirán un conocimiento común. Las clases evitan la duplicación inecesaria de código. Las funciones comunes de las funciones `ImprimeSubtotal()`, `ImprimeTotal()`, `ImprimeProducto()` las agrupan dentro de una clase que posea un conjunto de datos comunes a todas ellas. Dicho conjunto de conocimientos podría ser: el conjunto de códigos para cambiar el estilo de la letra, saltos de línea, conocer el estado de la impresora en cualquier instante dado, desplegar un mensaje de error si la impresora está fuera de linea, etc

La determinación de la responsabilidades correctas de una clase nos permite reutilizar el código. Por ejemplo, para la clase Imprime algunas de las responsabilidades que encontramos son necesarias para cualquier aplicación aunque esta no tenga que ver con terminales punto de venta.

III.1.3 Determinación de los métodos

Dentro de la POO, cada objeto es solo una instancia de clase. Tal como hemos visto, cada objeto posee un conjunto de conocimientos y responsabilidades que debe cumplir.

Un sistema orientado a objetos puede diseñarse en base a una *arquitectura cliente-servidor*. Esta filosofía, sugerida por Wirfs-Brock, establece que una clase debe convertirse en una servidora de los otras clases. Esta relación cliente-servidor se establece por medio de un contrato de servicio que se establece en el mensaje que un objeto manda a otro.

¿ Qué es una responsabilidad ? Las responsabilidades abarcan las dos partes constitutivas de un objeto. Por una parte, un objeto debe ser responsable de mantener la información que forma parte de su acervo de conocimiento. Por ejemplo, una clase que maneje el terminal de video debe ser capaz de informar a una clase cliente si el monitor se encuentra en modo gráfico o en algún modo de texto comprimido.

Por otra parte, las responsabilidades de una clase incluyen las acciones que este puede ejecutar. Se ha mencionado que dichas acciones proviene de la lista de requerimientos. Cada verbo en dicha lista es una acción que debe ser ejecutada por una clase, o por la interacción de diversas clases. Un servidor puede responder a un mensaje con la ejecución de una acción, o con una información producto de una acción.

Las responsabilidades de las clases pueden ser de dos tipos: las que corresponden a la interface pública de la clase y las que corresponden a la interface privada. Una interface pública está destinada a satisfacer los requerimientos de objetos ajenos a la clase. Ninguna clase externa debe ser capaz de accesar los datos o las responsabilidades privadas de una clase a la que le sea requerido algún servicio. Esto sería violar el principio de ocultamiento de la información y exponemos a que el acoplamiento entre los objetos sea tan fuerte que comience a causar los problemas que ya se han expuesto en el capítulo uno.

El diseño de la clase debe definir con extremo cuidado cual será la forma como trabajará la interface que establecerá el contrato de la clase servidora con el mundo exterior. Los objetos solo podrán comunicarse entre sí por medio de los mensajes que se envíen unos a otros.

IDENTIFICANDO LAS RESPONSABILIDADES

En primer instancia, una clase debe de responsabilizarse por los datos que conforman su conocimiento. El principio de diseño modular y el principio de ocultamiento de información insisten sobremanera en que las acciones y los conocimientos que se encuentren relacionados deben ir juntos.

Una clase debe convertirse en la administradora de sus datos. Ninguna clase que no sea ella misma debe tener acceso a su información privada. El diálogo de los objetos administradores es muy importante. Por ejemplo, un diálogo incorrecto es el siguiente

- ¿Está la impresora local lista?

- ¡Mmm! No, no esta lista

- ¿Está la impresora remota lista?

- ¡ Oh sí !

- Imprime el reporte de ventas

- Hecho

Un diálogo más correcto es el siguiente:

- ¿Hay alguna impresora lista?

- "¡Mmm! La impresora local no está lista, pero la impresora remota está libre.", *Si.*

- Imprime el reporte de ventas

-Hecho

El administrador de impresión tiene la responsabilidad de establecer en cual impresora debe ser enviado el reporte. Un objeto ajeno a la clase solo debe saber si es posible enviar el reporte o no. Una clase más elaborada podría simplemente emitir un mensaje ordenando la impresión del reporte, la clase administradora de la impresión podría entonces formar el reporte en una cola de impresión hasta que se encontrara libre alguna de las impresoras. De esta forma liberaría a la clase que emitió el mensaje de la responsabilidad de manejar una impresión fallida.

- Imprime el reporte de ventas

- "¡Mmm! La impresora local esta ocupada al igual que la impresora remota. Bien, colocaré la impresión en un archivo en disco y cuando se desocupe alguna de las impresoras enviare la impresión". *Hecho*

Por otra parte, las acciones que son propias de la clase son responsabilidad de la clase en cuestión. Confrontando las responsabilidades de las clases por medio de las tarjetas CRC se puede establecer la redundancia de responsabilidades entre las clases que conforman el sistema o la falta de ellas.

Cuando se trabaja en equipo en un diseño orientado a objetos, cada persona puede tomar algunas de las tarjetas CRC. Una vez que se han repartido las tarjetas se realiza una simulación de las responsabilidades que la tarjeta presenta. Una persona deberá leer el mensaje en voz alta solicitando un servicio. El resto de los diseñadores examinarán sus tarjetas para encontrar alguna que cumpla con dicha responsabilidad. Si dos objetos la pueden llevar a cabo significa que existe algun tipo de duplicación de funciones o que la interface no ha sido claramente definida. Es necesario realizar una validación al respecto. Por otra parte, si al ser emitido un mensaje, una persona encuentra que a ella le corresponde ejecutar una respuesta a dicha petición, pero no posee todos los datos necesarios es seña de que la clase podría no encontrarse correctamente definida.

Esta forma de simular el comportamiento de un sistema en por demás clarificante sobre las acciones que debe realizar cada clase y ayuda a las responsabilidades que no son visibles en primera instancia.

ASIGNANDO RESPONSABILIDADES

La asignación de las responsabilidades debe seguir algunas reglas que nos ayudarán a realizar mejores sistemas. Algunas de las más poderosas características de la POO se encuentran en la correcta asignación de responsabilidades y conductas que pueden ser heredadas a nuevos objetos. El nivel de abstracción para cada clase es de vital importancia en el diseño orientado a objetos para que se pueda reutilizar el código.

Las líneas generales que se sugieren para lograr una mejor asignación de las responsabilidades son:

- Distribuir la inteligencia del sistema: Cada clase y cada objeto dentro del sistema debe ser responsable de un conjunto bien delimitado de acciones y de un subconjunto del conocimiento total del sistema. Aunque se podría diseñar un sistema en donde una sola clase fuese responsable de cualquier acción a ejecutarse y poseedora de todo el conocimiento, la complejidad inherente sería muy grande. El mantenimiento implicaría una comprensión de una clase *todopoderosa y monolítica* con el consiguiente aumento en la compejidad para el programador de mantenimiento y aumento en el tiempo de corrección del error.

  Menor cantidad de inteligencia en un objeto, implicará que dicho objeto es más sencillo de implantar y de mantener.

- Establecer responsabilidades lo más general que sea posible: Cada objeto conoce algo de si mismo. Se puede pensar en un triángulo que sabe cuál es su área y la forma de obtenerla. Un círculo también puede saber como obtener su área. Triangulo y Circulo pueden ser dos objetos de un sistema CAD. La responsabilidad para obtener el área no debe de particularizarse en AreaDelTriangulo y AreaDelCirculo. La generalidad de la responsabilidad debe de ser definida como AreaDeLaFigura.

- Conservar las conductas con la información relacionada: Tal y como se había establecido en párrafos anteriores, la información debe estar localizada en los mismos objetos que realizan las acciones que la ocupan. El encapsulamiento de la información permite a cada clase salvaguardar su información para que no sea accesada por otras clases. Esto implica que el conocimiento está restringido a un poseedor. Cada objeto debe de disponer del conocimiento necesario para llevar a cado sus responsabilidades.

- Colocar la información acerca de una cosa en un solo lugar. Las clases deben convertirse en las administradoras de la información que contienen. Ningún otro objeto debe poder accesar dicha información. Si algún otro objeto requiere de información privada, la clase servidora deberá establecer una interface de comunicación de información con el exterior. Observar religiosamente el encapsulamiento de los objetos conduce a un menor acoplamiento entre ellos y a delimitar de mejor manera las responsabilidades.

  Compartir información puede llegar a constituir un problema al tener que mantener datos redundantes. La información debe mantener *n* copias de sí misma, cada vez que esta sea modificada, será necesario modificarla en todas las partes en donde exista una copia de la misma. Centralizar la información es una alternativa mucho más aceptable.

- Compartir responsabilides. Una clase no siempre puede realizar una actividad por sí misma. En general podemos decir que se auxilia de clases más especializadas. O bien, las conductas de una clase pueden estar definidas por su relación con clases que las contienen. Mencionabamos el caso de los gatos. Estos pertenecen a una superclase que los contiene y que determina parte de sus características. Así que los gatos, al ser carnívoros, poseen las características de todos los carnívoros y además las características que los diferencian como los felinos egoistas y vanidosos que son.

El diseño de responsabilidades requiere de una estrecha colaboración entre todos los objetos del sistema.

III.1.4 Escritura del programa principal

Este capítulo no pretende enseñar en forma profunda el lenguaje C++. Solo se mostrarán los aspectos más relevantes de dicho lenguaje para implantar programas orientados a objetos.

CLASE IOSTREAM

El lenguaje C++ posee una serie de características que los hacen distinto de el lenguaje C estudiado hasta este momento. La primera que estudiaremos serán las clases de entrada y salida predefinidas para C++.

Al igual que en el lenguaje C procedimental, C++ no posee una función determinada para las salidas y las entradas, en su lugar hace uso de la *corriente* ó *secuencia* de salida (*stream*: corriente. Alusión a la corriente de datos generada en una operación de entrada y salida). En lugar de la función `printf()` y su contraparte `scanf()` se hace uso de los operadores << y >>.

El operador << se utiliza asociado con el archivo de cabecera `iostream.h`. En general se encuentra asociado a un dispositivo de almacenamiento temporal conocido como *cout* . Este operador despliega en el dispositivo de salida los argumentos que se coloquen a su derecha.

Ejemplo:
Realizar un programa que despliege una cadena de caracteres, un número real y un entero.

```
// Utilizacion del operador << para desplegar datos
// 29/mar/1993

#include <iostream.h>
#define SALTO '\n'

main()
{
   int entero =2;
   double real =5.4;
   char mensaje[] = "Este es un ejemplo de salida: ";

   cout  << mensaje;
   cout << "entero = "<< entero << SALTO;
   cout << "real= " << real << SALTO;
}
```

Resultados:

```
Este es un ejemplo de salida: entero = 2
real= 5.4
```

Observaciones:

El operador << no requiere de especificadores de campo para el despliege de datos, inclusive, puede mezclar distintos de datos en una sola secuencia de salida. El macro `SALTO` nos define el salto de linea que se requiere.

El operador >> proporciona las entradas necesarias para el programa. El operador >> se encuentra asociado al dispositivo de entrada *cin* . Al igual que con el operador <<, es necesario utilizar el archivo de cabecera `iostream.h` en donde se encuentra la definición de ambos operadores.

Las flechas indican el sentido en el cual se mueven los datos, de tal forma que con el operador << se apunta al dispositivo *cout*. Para el operador >> las flechas salen de *cin*.

Ejemplo:

```
// Ejemplo de uso del operador >>
// 29/mar/93

#include <iostream.h>

main ()
{
int cantidad;
float precio;
char *mensaje = "Digite cantidad y precio unitario:  ";
cout << mensaje << flush;
cin >> cantidad >> precio;
cout << cantidad << " unidades a " << precio << " costarán : ";
cout << (cantidad * precio ) << endl;
}
```

Resultados:

```
Digite cantidad y precio unitario:  5
12.50
5 unidades a 12.5 costarán : 62.5
```

Observaciones:

Al igual que con el operador de salida, la entrada acepta distintos tipos de datos sin necesidad de especificar un formato de entrada para cada uno de los datos. En la primer línea que envía en menasje a cout aparece la palabra `flush` la cual fuerza a cout a vaciar su contenido en el dispositivo de salida. Así mismo, se utiliza la palabra `endl` para indicar un salto de linea.

El operador de entrada es lo suficientemente versátil como para concatener una serie de entradas, tal y como lo hace el operador de salida. Esto convierte a estos operadores en excelentes alternativas a `scanf()` y `printf()`. Además, tal y como se verá en posteriores apartados, es posible definir entradas y salidas para datos creados por el usuario.

La clase iostream define cuales son los principales medios para realizar entradas y salidas. Sus operadores puedan ser sobrecargados. Se explicará ampliamente lo que esto significa en el apartado III.7.

ARGUMENTOS POR OMISION

En el lenguaje C no es posible dejar de incluir un argumento dentro de una lista de parámetros. En C++ es posible definir una serie de valores por omisión. Es decir, si el usuario de una función omite enviar un valor de parámetro, la función asignará un valor predefinido a dicho argumento.

Ejemplo:

```
// Argumentos por omisión
// 29/mar/93

#include <iostream.h>
#ifndef GRAFO
      #define GRAFO '*'
#endif

void pintaestrellas(int limite=8)
{
      register int contador;

      for (contador = 0 ; contador < limite; contador ++)
            cout << GRAFO;
}

void main(void)
{
      cout<< "Se pintarán 20 estrellas"<< endl;
      pintaestrellas(20);
      cout<< endl;
      cout << "Se pintara un valor por omisión "<<endl;
      pintaestrellas();
}
```

Resultados:

```
Se pintarán 20 estrellas
********************
Se pintara un valor por omisión
********
```

Observaciones:

La función `pintaestrellas()` desplegará tantos asteriscos como lo indique un argumento pasado por el usuario de la función. En la primer llamada realizada en `main()` se informa a la función que debe desplegar 20 asteriscos. Sin embargo en la segunda llamada no se especifica ningún número. La funcion decide entonces que deberá tomar el argumento predefinido ya que se ha omitido un valor por parte de la llamada. Asigna el valor de ocho al límite de asteriscos a desplegar y ejecuta la función.

En una función pueden definirse argumentos que requieran ser especificados, junto con argumentos que tomen valores por omisión. Se pueden tener tantos argumentos por omisión como sea necesarios, pero si se han de mezclar con argumentos que necesitan ser definidos, estos siempre encabezarán la lista.

Ejemplo:

```
// Argumentos por omision
// Mezcla de argumentos
// 29/mar/93

#include <iostream.h>
#ifndef GRAFO
      #define GRAFO '*'
#endif

void pintacuadro(int ancho,int largo=8)
{
      register int renglon;
      register int columna;

      cout << endl;
      for (renglon = 0; renglon < ancho; renglon ++)
      {
          for (columna = 0 ; columna < largo; columna ++)
            cout << GRAFO;
          cout << endl;
      }
}

void main(void)
{
      cout<< "Se despliega cuadro de 10 x 5"<< endl;
      pintacuadro(10,5);
      cout<< endl;
      cout << "Se despliega cuadro de 5 x valor por omision "<<endl;
      pintacuadro(5);
}
```

Resultados:

```
Se despliega cuadro de 10 x 5

*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

```
Se despliega cuadro de 5 x valor por omision

********
********
********
********
********
```

Observaciones:

Se puede observar que en la primer llamada a la función **pintacuadro()** se especifican los dos argumentos, y la función responde a ellos. Sin embargo, en la segunda llamada solo es especificado uno de los parámetros necesarios, al segundo de ellos es asignado el valor por omisión. No es posible hacer una llamada a la función **pintacuadro()** sin argumentos.

FUNCIONES EN LINEA (*INLINE*)

Una función en linea funciona de la misma forma que una macro del preprocesador. El código definido dentro de una especificación en linea es repetido en todas las referencias a dicha llamada. El compilador trata a una función en linea como una función, de tal forma que los parámetros de una función no deben ser tratados con tanto cuidado que como en una macro. Esto es muy benéfico porque existen menos riegos de equivocación en los programas por comportamiento inesperado.

Ejemplo:

```
// Funciones en linea
// 29 /mar/1993

// #include <stdio.h>
#include <iostream.h>

inline long int cubo(int x = 1)
{
      return (x*x*x);
}

main()
{
      cout << "El cubo de 9 es: "<< cubo(9) << endl;
      cout << "El cubo de 5+4 es: " << cubo (5+4) << endl;
}
```

Resultados:

```
El cubo de 9 es: 729
El cubo de 5+4 es: 729
```

Observaciones:

La función en linea se especifica a través de la palabra reservada **inline**.

Como puede apreciarse del programa ejemplo, la función en linea trabaja como cualquier función escrita dentro del programa, con la excepción del tratamiento que recibe por parte del compilador. Una macro definida exactamente para ejecutar el cubo de un número entero habría tenido que llevar los paréntesis para establecer correctamente la prioridad de evaluación.

Las funciones en linea son recomendables solamente en aquellos casos en los que la función sea de pequeñas dimensiones. Si la función en linea es muy grande, la cantidad de código total en el programa crece considerablemente y por tanto disminuye la eficiencia.

ARGUMENTOS POR REFERENCIA

El apartado III.4 estuvo dedicado al manejo de los apuntadores como elementos que nos permiten manipular las direcciones en las cuales se alojan las variables de un programa. En el apartado II.4.4 se explicó la forma de utilizar los apuntadores como argumentos a funciones.

C++ utiliza un nuevo tipo de paso de parámetros. Observemos el siguiente ejemplo:

```
// Argumentos por referencia
// 23/mar/1993
#include <iostream.h>

int cuad(int& variable)
{
      variable=variable*variable;
      return(variable);
}

void main( void)
{
      int valor = 3;
      cout << "El cuadrado de " << valor << " es:";
      cuad(valor);
      cout << valor << endl;
}
```

Resultados:

```
El cuadrado de 3 es:9
```

Observaciones:

La función `cuad()` obtiene el cuadrado de un número y almacena el resultado en la misma variable. La función define el argumento a través de & con lo cual se indica un paso de parámetros por referencia. Cuando la variable `valor` es desplegada en la función `main()` se observa que su contenido fue modificado.

Es muy importante remarcar la claridad del código de la función `cuad()`. Observese la siguiente comparación entre los tres métodos para pasar parámetros:

Ejemplo:

```
// Argumentos por referencia
//23/mar/1993
#include <iostream.h>

int &cuadreferencia(int& variable)
{
      variable=variable*variable;
      return(variable);
}

int cuadvalor( int variable)
{
      variable=variable*variable;
      return(variable);
}

int *cuadpuntero(int* variable)
{
      *variable = *variable * (*variable);
      return(variable);
}

void main( void)
{
      int valor = 3;

      // LLamada por valor
      cout << "El cuadrado de " << valor << " es:";
      cuadvalor(valor);
      cout << valor << endl;

      // Llamada por referencia
      cout << "El cuadrado de " << valor << " es:";
      cuadreferencia(valor);
      cout << valor << endl;

      //Llamada por puntero
      cout << "El cuadrado de " << valor << " es:";
      cuadpuntero(&valor);
      cout << valor << endl;

}
```

Resultados:

```
El cuadrado de 3 es:3
El cuadrado de 3 es:9
El cuadrado de 9 es:81
```

Observaciones:

La primera función que encontramos es una llamada por valor, en ella aunque la función altera el valor del argumento , esto no se ve reflejado en la variable del programa principal.

La segunda llamada es una llamada por referencia en donde la variable del módulo padre sí es modificada por la función `cuadreferencia()`. El código de esta función es prácticamente tan claro como el de la llamada por valor.

La última llamada es realizada por medio de una variable puntero. El código es mucho más complejo y de difícil lectura. El programador se ve forzado a preocuparse por lo que el código hace, en lugar de preocuparse por lo que el código dice.

Aunque los punteros solucionan la mayor parte de los problemas que se presentan, existe una aplicación específica dentro de la POO en C++ que hace necesario el uso de los parámetros por referencia. Esto se verá con mayor amplitud en el apartado III.7

III.1.5 Determinación de los elementos

Antes de iniciar con una la definición de clases en C++, hagamos un provechoso repaso de las estructuras.

Una estructura en C es una colección de datos que se agrupan bajo un solo nombre. Cada dato es independiente de los demás e inclusive puede ser de distinto tipo. Por ejemplo, una estructura conteniendo los datos de un alumno, puede definirse de la siguiente manera:

```
struct pupilo
{
      char nombre[20];
      char a_paterno[15];
      char a_materno[15];
      char num_cuenta[7]
      unsigned int edad;
}
```

DEFINICION DE UNA CLASE

Una clase puede ser visto como un objeto. Posee información, y conoce los métodos con los cuales realiza su trabajo. Una estructura es una colecccción de elementos agrupados como un dato compuesto más abstracto. Ningún miembro de una estructura puede ser accesado individualmente sin referirse a la estructura de la cual forma parte.

Una clase se define de una manera muy parecida a como se declara una estructura, observese el siguiente ejemplo:

```
// Definicion de una clase
// 23 / marzo / 1993
```

```
class examen
      {

            // Estos miembros son vistos por cualquier objeto
            public:

                  char alumno[50];
                  char cve_tipo[3];

                  void asigna_calificacion(int& calificacion);
                  int Obten_calificacion(void);

            // Estos miembros solo pueden ser accesados por miembros de la
            // misma clase
            private:
                  int evaluacion;
                  int num_preguntas;

                  int promedio (int num_preguntas, int& evaluacion);
                  void respuesta(int num_preguntas, int respuesta);
      };


//  Programa principal
void main(void)
{
      // Con el nombre de la clase se definen objetos o instancias de clase

      examen parcial, final;
}
```

Una estructura conoce los elementos que la conforman. Sin embargo no tiene control sobre las funciones que manipularán a sus miembros. Una clase define los datos que conforman el conocimiento que tiene de sí misma y define los métodos que podrán manipular a los datos de la clase. Una estructura solo posee datos. Una clase posee datos y funciones como miembros.

En ejemplo se tiene que cada examen debe tener el nombre del alumno que lo sustenta, así como una clave que se le asigna a cada examen. A su vez, cada examen debe ser capaz de asignar una calificación y de daría a petición expresa de algún objeto que la requiera. Estos son datos que todos los objetos pueden y deben conocer.

ACCESO A LOS MIEMBROS DE UNA CLASE

La palabra reservada public indica que los datos y los métodos declarados a su continuación son de tipo publico y por tanto, cualquier objeto puede accesarlos.

Notamos también que en la definición de la clase se encuentran el número de preguntas de las que consta el exámen, el método para determinar el promedio y la función que acepta las respuestas del alumno. Estos miembros de la clase son conocidos solo por los miembros de esa misma clase. Ningún otro objeto perteneciente a una clase distinta puede accesar estos miembros. Esta características convierte a los miembros en privados.

La palabra reservada **private** indica que los datos y los métodos declarados a su continuación son de tipo privado y por tanto solo los miembros de la misma clase pueden accesarlos. Esto es lo que hace posible el ocultación de información, escencial en la programación orientada a objetos.

En tanto que en una estructura todos los miembros son públicos, en una clase los miembros son privados por omisión. El encapsulamiento es una parte medular de la POO es por ello que, si se omiten las palabras **private** o **public**, los miembros serán definidos como privados.

Como una buena práctica de programación se siguiere que los elementos públicos de una clase se definan justo al principio de la declaración de clase. Cuando otros programadores se encuentren buscando clases que les puedan ayudar a solucionar sus problemas, lo que estarán buscando es la forma de usar una clase. Es decir, estarán buscando la interface pública de la clase. Solo en aquellos casos en los cuales se tenga un genuino interés por saber como se implanta una clase, leerá la parte privada de una clase.

En los programas que utilizan funciones prototipos, nosotros definiamos una sección para la declaración de las funciones (funciones prototipo) y una sección separada para definir las funciones .El código fuente de una clase se suele escribir también en dos partes. Estas partes se escriben en dos archivos separados. En uno se escriben las definiciones de clase sin detallar las implantaciones. El equivalente en la programación tradicional sería escribir las funciones prototipo en un archivo de cabecera con extensión .h. La implantación de las funciones se realiza en un archivo separado. Finalmente estos archivos se compilan en el archivo del programa que los ocupe haciendo uso de la directiva `#include`.

Es muy importante recordar los conceptos que se expusieron en el capítulo uno sobre **El concepto de caja negra** (vease I.1.2). Todos esos conceptos siguen siendo válidos.

La forma general para la definición de una clase es:

```
class nombre_clase
{
public:
        tipo variable ;
        tipo variable ;

        tipo nombre_función (tipo variable, ...);
        tipo nombre_función (tipo variable, ...)
        {
                bloque de sentencias
        }
protected:
        tipo variable ;
        tipo variable ;

        tipo nombre_función (tipo variable, ...);
        tipo nombre_función (tipo variable, ...)
        {
                bloque de sentencias
        }
```

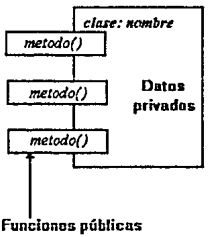
```
private:
        tipo variable ;
        tipo variable ;

        tipo nombre_función (tipo variable, ...);
        tipo nombre_función (tipo variable, ...)
        {
                bloque de sentencias
        }
}
```

La declaración está conformada por tres secciones. Aunque no es necesario que existan las tres clases. La variables y las funciones miembro son **private** por omisión. La declaración de las funciones miembros puede reducirse a la función prototipo. Cada sección define el acceso que poseerán los datos encerrados entre las palabras reservadas **public, private y protected.**

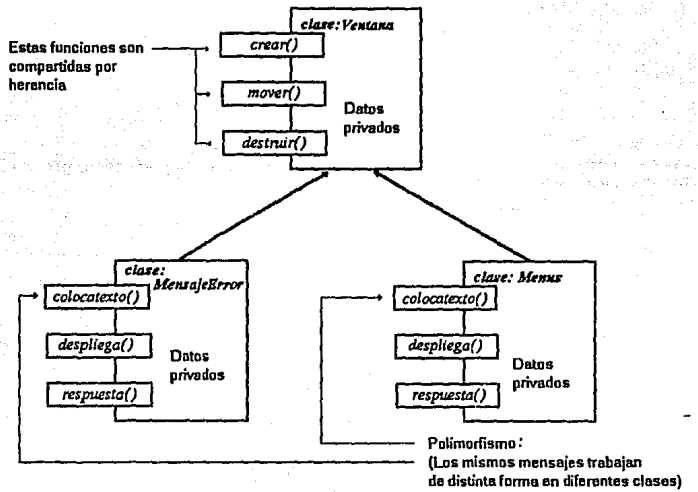
A reserva de ampliar el tema cuando se trate el tema de herencia (ver III.2) los datos de tipo protegido son accesibles a las clases derivadas de esta.

NOTACION GRÁFICA

Aunque aún no exista un estándar en cuanto a la manera de representar un objeto, la notación propuesta por Booch está teniendo bastante aceptación dentro de los medios de diseño orientado al objeto. Esta consiste de un cuadro que representa al objeto o a la clase que se esté trabajando. Dentro de dicho cuadro se encapsulan los datos y las funciones que conforman a la entidad representada. Las funciones que conforman la interface pública del objeto se enmarcan en rectángulos que sobresalen al cuadro principal.



Los objetos también se relacionan entre sí en forma jerarquica. Cuando esto sucede algunas de sus características son heredadas a las clases inferiores.



Un objeto que no forma parte de la clase no puede accesar a los datos de otra clase. Esto es muy importante de tomar en cuenta en el proceso de diseño de una clase. No es conveniente que los programadores puedan accesar a los datos que son escencialmente privados a una clase determinada. El buen diseño de las interfaces públicas permitirá la correcta implantación y la reutilización de las clases.

ARCHIVOS DE CABECERA

La definición de las clases es conveniente tenerla en archivos de cabecera que serán compilados en forma separada utilizando la directiva `#include`. Con esto podremos reutilizar las definiciones en cada nuevo sistema que las requiera. La implantación de los métodos se realizará en un archivo separado. Se insiste en la necesidad de proporcionar una interface de rápida lectura para los programadores que deseen reutilizar una clase.

Los archivos de cabecera tienen la extensión .h, .hxx o .hpp. Nosotros usaremos la extensión .h. Los archivos en donde se implantan los métodos suelen tener la extensión .c, .cxx o .cpp: Nosotros usaremos la extensión .cpp

III.1.6 Implantación de los métodos

Los métodos son las funciones a través de las cuales los objetos pueden ejecutar sus responsabilidades.

Una función puede definirse dentro del cuerpo de la definición de clase. Cuando esto sucede el compilador asume que la definición de la función es en línea. Es decir, que en cada referencia que se encuentre dentro del programa fuente, se realizará una sustitución del código de la función.

Cuando la función se define fuera de la declaración de una clase se asume que es una función como cualquier otra en lenguaje C.

Una función miembro en línea debe contener poco código para que sea realmente eficiente, ya que cada referencia aumentará la extensión del código.

SENTENCIAS NEW Y DELETE

El lenguaje C++ dispone de dos nuevas sentencias. Estas nos sirven para crear y destruir objetos .

Cuando estudiamos el alojamiento de estructuras bajo asignación dinámica de memoria, utilizabamos la función `malloc()` par realizar una petición de memoria. Dicha petición tenía una forma general muy semejante a la siguiente:

```
struct ejemplo
{
      int dato1;
      float dato2;
}
struct ejemplo *Puntero_Estruct;
Puntero_Estruct = (struct ejemplo *) malloc( sizeof (struct ejemplo) );
```

En el lenguaje C++ este código se puede abreviar de la siguiente manera:

```
ejemplo *Puntero_Estruct;
Puntero_Estruct = new ejemplo
```

A su vez, para liberar la memoria obtenida a través de la función malloc() se utilizaba la función free(). En C++ podemos utilizar el operador delete de la siguiente manera:

```
delete Puntero_Estruct;
```

USO DE UNA CLASE

Para hacer uso de una clase es necesario declarar instancias de dicha clase. La declaración de instancias se realiza utilizando el nombre de la clase y el nombre del objeto que deseemos crear. Por ejemplo, cuando nosotros declaramos un dato de tipo entero estamos diciendo que la variable es una instancia de la clase enteros:

int contador;

De la misma manera, si se define una clase con el nombre de autos, para definir instancias de clase bastará con la siguiente sentencia:

autos vochito, renolio;

Cuando deseamos que una persona ejecute una determinada acción nosotros decimos algo como esto:

- *Humberto, escribe tu nombre*
- *Olga, mira la televisión*

Estas ordenes están constituidas por: el nombre de la persona responsable (Humberto, Olga), la acción que deben ejecutar (escribir, mirar) y el objeto con el cual se actuará (nombre, televisión).

De la misma forma para mandar un mensaje a un objeto, primero escribimos el nombre del objeto, separado con un punto, la función miembro que define la acción que deseamos que se ejecute y como argumento de la función, el dato sobre el cual se opera.

Supongamos que se implanta una clase llamada **Persona** que define el método para que un objeto escriba su nombre, entonces la orden verbal vista hace un momento podría escribirse en C++ de la siguiente manera:

```
// Definimos el nombre
char *nombre = "Humberto San";

// Con el nombre de la clase definimos una instancia
Persona Humberto(nombre);

//   -Humberto, escribe tu nombre
Humberto.escribir(nombre);
```

Aunque al principio parezca extraña esta notación es posible acostumbrarse con rapidez cuando la llamada se lee exactamente como lo indica el comentario. Debe recordarse que ahora la función es un miembro de una clase. El punto es un indicativo de esta relación de pertenencia

Ejemplo:
Implantaremos una clase que cree una mensaje con funciones en linea, y lo despliege en pantalla.

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


class mensaje
{
      public:
            // Funciones en linea
            // Esta funcion miembro crea una cadena
            mensaje(const char *mensg)
            {
                  int longitud;
                  longitud = strlen (mensg);
                  punt_cad = new char[longitud];
                  strcpy(punt_cad,mensg);
            }
```

```
            // Destruccion de la cadena
            ~mensaje()
            {
                  delete punt_cad;
            }

            // Definicion de una funcion miembro
            // Despliega el contenido de la cadena
            void despliega(void)
            {
                  cout << punt_cad << endl;
            };

            // Dato privado de la clase
            //Puntero a la cadena creada
            private:
            char *punt_cad;
      };

void main(void)
{
      char *saludo = "¡ Hola alumnos de C++ !";
      mensaje Hola(saludo);
      mensaje Adios("¡Esto fue todo amigos!");

      Hola.despliega();
      Adios.despliega();
}
```

Resultados:

```
¡ Hola alumnos de C++ !
¡Esto fue todo amigos!
```

Observaciones:

La lectura de este programa puede no resultar tan evidente la primera vez que se trabaja con programas orientados a objetos. Así que explicaremos paso a paso su funcionamiento.

Cuando `main()` inicia su ejecución declara un puntero (`saludo`) a una cadena que contiene el mensaje "`¡ Hola alumnos de C++ !`". En la siguiente linea se declara una instancia de la clase mensaje. Dicha instancia recibe el nombre de `Hola`.

En la definición de la clase se tiene una función que tiene el mismo nombre que la clase. A dicha función se le conoce con el nombre de *constructor* . Esta función se activa automáticamente cada vez que es creado un objeto. El constructor de la clase mensaje recibe un puntero a una cadena y lo asigna a un puntero que será propiedad del nuevo objeto. El puntero `punt_cad` forma parte del objeto `Hola`. El objeto es ahora poseedor de un mensaje y puede ejecutar todas las funciones de su propia clase.

El puntero `Hola.punt_cad` apunta al mensaje "`¡ Hola alumnos de C++ !`". Observese que para referirnos al dato utilizamos el operador miembro para separar el nombre del objeto y el nombre del dato.

Tan pronto como la asignación de la cadena asociada al objeto **Hola** es realizada, la función **main()** encuentra una nueva declaración. Esta vez se declara el objeto **Adios**. Todos las variables públicas o privadas de la clase son copiadas para que este objeto pueda almacenar sus propios datos. El constructor se encarga de establecer un nuevo puntero para el mensaje de este objeto.

El puntero **Adios.punt_cad** apunta al mensaje **"¡Esto fue todo amigos!"**. La función **main()** no puede accesar este puntero pues no pertenece a la clase.

La función **despliega()** fué definida como pública, por tanto la función **main()** puede accesarla para pedir a cada uno de los objetos que vacien sus mensajes sobre el dispositivo conectado a cout. Cuando el programa encuentra la instrucción **Hola.despliega()** sabe que el objeto forma parte de la clase **mensaje** por lo que esto es equivalente a decir *llama a la funcion despliega de la clase mensaje usando la información del objeto Hola.* Dicho en otras palabras se realiza una llamada a **mensaje.despliega()** y se utiliza el dato **Hola.punt_cad**

La segunda llamada de despliege es realizada por el objeto **Adios** por lo que se realiza una llamada a **mensaje.despliega()** usando **Adios.punt_cad.**

FUNCIONES FUERA DEL CUERPO PRINCIPAL

Todas las funciones de esta clase han sido definidas en linea ya que se encuentran dentro de las llaves de la declaración de la clase. Una forma alterna de definir esta misma clase con las funciones fuera de los paréntesis es la siguiente:

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


class mensaje
{
      public:
            // Funciones en linea

            mensaje(const char *mensg);
            ~mensaje();
            void despliega(void);

            // Dato privado de la clase
            //Puntero a la cadena creada
            private:
            char *punt_cad;
};

// Esta funcion miembro crea una cadena
inline mensaje::mensaje(const char *mensg)
{
      int longitud;
      longitud = strlen (mensg);
      punt_cad = new char[longitud];
      strcpy(punt_cad,mensg);
}
```

```
// Destruccion de la cadena
inline mensaje::~mensaje()
{
      delete punt_cad;
}
// Definicion de una funcion miembro
// Despliega el contenido de la cadena
inline      void mensaje::despliega(void)
{
      cout << punt_cad << endl;
};


// Funcion principal

void main(void)
{
      char *saludo = "¡ Hola alumnos de C++ !";
      mensaje Hola(saludo);
      mensaje Adios("¡Esto fue todo amigos!");

      Hola.despliega();
      Adios.despliega();
}
```

La sentencia inline indica al compilador que aunque la función se encuentra fuera de los paréntesis de la declaración de clase, deberá ser considerada como en linea.

Una nueva característica es el operador de resolución de ámbito `::`. Este operador indica al compilador que la función que se define pertenece a una clase determinada. Su forma general es:

nombre_clase :: nombre_funcion_miembro

Con este operador se evitan ambiguedades en la definición de las funciones miembro. Mas de una clase puede tener funciones miembros con el mismo nombre `leer()`, `escribir()`, etc. Con la especificación de la clase a la cual pertenecen se aclara el alcance de cada función.

Veamos otro ejemplo de una clase:

```
// Funciones miembro y resolucion de ambito
// 30/mar/1993

#include <iostream.h>
#include <conio.h>


class ejemplo
      {
            public:
            void pintacuadro(int renglon, int columna);
            private:
            void pintaasteriscos (int limite);

      };
```

```
void ejemplo::pintaasteriscos (int limite)
            {
                  int contador;
                  for (contador = 0 ; contador < limite; contador ++)
                  cout << '*';
            }

void ejemplo::pintacuadro(int renglon, int columna)
            {
                  int contador;
                  for (contador = 0; contador < renglon; contador++)
                  {
                        pintaasteriscos(columna);
                        cout << endl;
                  }
            }
void main (void)
{
ejemplo A;
clrscr();
A.pintacuadro(5,5);
/*
A.pintaasteriscos(10) es incorrecto en main()
*/
}
```

Resultados:

```
*****
*****
*****
*****
*****
```

Este ejemplo muestra la forma como se declaran funciones de carácter público o privado. La función `main()` solo puede accesar a la función `ejemplo::pintacuadro()`. La función `ejemplo::pintaasteriscos()` solo es posible accesarla a través de los miembros de la clase ejemplo. Si la función `main()` quisiera accesar a dicha función se detectaría un error en tiempo de compilación debido a que se trata de una función oculta para todos los objetos ajenos a la clase.

ESTADOS E IDENTIDAD DE UN OBJETO

Un objeto no solo sabe como hacer su trabajo, también posee determinados conocimientos que lo hacen una instancia distinta a todas las demás de su misma clase y determinan su identidad. Un objeto posee un estado determinado que va cambiando conforme el programa se ejecuta. En la programación tradicional, una función siempre inicia de la misma forma, inicializa los datos de la misma forma y, en pocas palabras, no puede encontrarse en un estado determinado.

La POO hace posible, a través del ocultamiento de la información que se pueda escribir un programa interesante como el siguiente:

Ejemplo:

Podemos imaginar que existen dos personas con estado de ánimo distinto. Uno está muy triste y la otra no. Una de las personas llega con la otra e inicia una conversación típica. Escribir un programa en donde dos objetos conversen sobre sus estados de ánimo. El programa terminará cuando ambos objetos se encuentren en un estado de ánimo tal que impida continuar la conversación. O sea ambos tristes.

```
// Conversacion entre dos objetos
// 30/marzo/1993
// Ernesto Peñaloza Romero


// Los objetos no solo poseen metodos
// también conocen su propio estado

#include <iostream.h>
#include <conio.h>

// Estados posibles para esta conversacion
#define yaestaba      0
#define llegando      1
#define triste        2
#define alegre        3
#define despidiendose 4

class conversador
{
      public:
            int estado;

            int platica(int humor);
            conversador(int estado);

            //Mi propio estado de animo
      private:
            int animo;
};

// constructor de clase

conversador::conversador ( int estado = yaestaba)
{
      if (estado == yaestaba)
            animo = triste;
      else
            animo = alegre;
}

//Este metodo es una patron de platica para
//cada instancia de la clase
// El parámetro que se recibe es el humor del otro objeto
// en base al cual se reacciona
```

```
int conversador::platica ( int humor)
{

      cout <<endl;
      // Actua en base al estado del otro
      switch (humor)
      {
            case yaestaba:
                  // Como el otro ya estaba, lo saludo con cortesia
                  cout << "¡ Hola me da gusto verte !";
                  break;
            case llegando:
                  // Si el otro está llegando, le saludo
                  cout << "¡ Hola ! ";
                  break;
            case triste:
                  // Si el otro está triste. Mejor me voy

                  cout << "¡ Oh ! Es una pena";
                  cout << endl << "No sabes cuanto me entristece";
                  estado = despidiendose;
                  animo = triste;
                  break;
            case alegre:
                  // Si el otro esta alegre me "alegro"
                  cout << "¡ Es bueno saber que alguien es dichoso!";
                  break;
      }

      cout << endl;

      if ( estado != despidiendose)
      {
            // Informa al otro sobre mi estado de ánimo
            cout << "Me encuentro ";
            switch(animo)
            {
                  case triste:
                        cout << "triste.";
                        break;
                  case alegre:
                        cout << "alegre.";
                        break;
            }
            // Elaboro la pregunta para continuar la conversación
            cout << endl << "¿ Qué tal?";
      }
      else
      {
            // Mensaje de despedida. No mas conversacion
            cout << " ¡Bueno nos vemos despues!";
      }
      cout << endl;
      return(animo);
}
```

```
void main(void)
{
      // Se crean dos instancias, cada uno en un estado distinto
      conversador Ruben(yaestaba);
      conversador Raul(llegando);

      int animo_otro;

      clrscr();
      while (! ( Raul.estado == despidiendose && Ruben.estado == despidiendose))
      {
            cout <<endl<< "Raul: ";
            animo_otro=Raul.platica(animo_otro);
            getch();
            cout << endl << "Ruben: ";
            animo_otro=Ruben.platica(animo_otro);
            getch();

      }
}
```

Resultados:

```
Raul:
¡ Hola me da gusto verte !
Me encuentro alegre.
¿ Qué tal?

Ruben:
¡ Es bueno saber que alguien es dichoso!
Me encuentro triste.
¿ Qué tal?

Raul:
¡ Oh ! Es una pena
No sabes cuanto me entristece
 ¡Bueno nos vemos despues!

Ruben:
¡ Oh ! Es una pena
No sabes cuanto me entristece
 ¡Bueno nos vemos despues!
```

Observaciones:

El constructor de la clase establece un estado de ánimo para cada uno de los objetos. Para que esta conversación sea posible, ambos objetos deben encontrarse en un estado de ánimo distinto o ambos tristes.

Ambos objetos tendran la misma conducta, independientemente de cual de los dos objetos sea el que se encuentre triste, finalmente la conversación terminará con la depresión de ambos. Los objetos no sabrían que hacer si ambos estuviesen alegres. Comenzarían una conversación sin fin en donde cada uno de ellos le comunicaría al otro su alegría.

Por supuesto que sería terrible si todas personas reaccionarán así: Los métodos para estos objetos los hacen huir de los objetos tristes. Si es detectada tristeza en el otro, de inmediato se ordena la despedida para evitar el contagio del mal. ¡Son bastante egoístas estos objetos!. Pero muestran con claridad que cada objeto posee su propio conocimiento sobre el estado que guardan las variables que lo conforman. Y que cada vez que se entra a una función miembro, ese estado es conservado. Los objetos poseen memoria porque poseen una copia *personal* de todas las variables que su clase les ha conferido.

IMPLANTACIÓN DE LOS MÉTODOS Y PROGRAMACIÓN ESTRUCTURADA

Un objeto es por tanto un conjunto de funciones y el conocimiento común que esas funciones poseen. La pregunta inevitable en este punto es: ¿La teoría de objetos expuesta hasta este momento hace que en implantación de los métodos sea imposible escribirlos bajo una metodología estructurada?

En este momento es claro que ya no es posible escribir programas orientados a objetos sin realizar un diseño previo sobre las responsabilidades que estos deben cumplir. Las tarjetas CRC serán la gran herramienta que nos permita realizar dicho diseño.

Sin embargo la implantación de los métodos, que no son más que funciones miembro de una clase, no se requiere de mayor conocimiento para su implantación que la lista de especificaciones sobre las tareas que dicha función deba realizar y la estructura de los mensajes que recibirá para poder realizar su trabajo. Un método es un programa, y su diseño debe implicar una metodología. Si bien este es un *"trabajo menor"* dentro de la POO, esto no implica que deba dejarse al azar.

Reiteramos nuestro punto de vista: la POO es una superconjunto que abarca a los conceptos en los que se funda el análisis y la programación estructurada. Existen autores que han propuesto métodos de análisis orientados a objetos basados en el análisis estructurado. Nosotros no vamos tan lejos. Simplemente abogamos por una escritura de los métodos basados en la programación estructurada.

Cada vez que surge una nueva técnica o una nueva filosofía, se tiende a pensar que constituye una negación de todo lo que le ha antecedido en el tiempo. Nada tan falso como esto. Aún la POO, es solo una evolución del pensamiento orientado a los datos y su conjunción con los modelos procedimentales.

Un método por tanto deberá de ser un programa propio en sí mismo, ser construido con las estructuras de control disciplinadas que se estudiaron en el capítulo uno y por sobre todas las cosas: debe ser una caja negra que cuide en todo momento del acoplamiento con los otros miembros de su clase, de las clases derivadas y de los elementos fuera de su misma clase. C++ provee de palabras reservadas para permitir al diseñador un acceso planeado a cada uno de los métodos.

Debe de pensarse que una clase puede constituir un sistema en sí mismo. Cada clase será propietaria de un conjunto de funciones que se encontrarán interrelacionadas. Cada clase poseerá áreas de conocimiento común. Si las interfaces dentro de la clase no se atienden con el debido cuidado, podría resultar tan ineficiente programar en un lenguaje orientado a objetos como programar inestructuradamente.

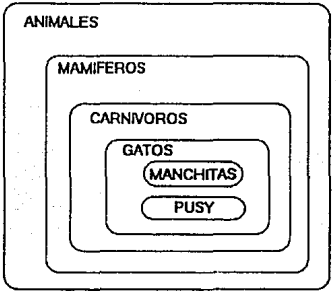
Con riesgo de parecer demasiado reiterativo. Cada método debe cumplir con una responsabilidad bien delimitada. En la POO, más que en cualquier otra disciplina, la función de cada módulo, debe ser cuidadosamente diseñada. La cohesión de cada método debe ser funcional. La cohesión de cada clase debe ser informacional. Se debe diseñar cada clase pensando en la posibilidad de reutilizarla en otros sistemas, sea por vía de simple uso o por vía de los mecanismos de herencia que estudiaremos en la próxima sección.

III.2 Clases, objetos y herencia.

Cuando clasificamos a un objeto como miembro de una clase estamos estableciendo que el objeto posee una serie de características propias de su clase.

Si en una visita a Juancito, este nos pidiera que alimentaramos a Manchitas, nosotros podríamos hacerlo en base a una serie de características que sabemos que posee, por el mero detalle de que se trata de un miembro de la clase de los gatos. Por ejemplo, sabemos que un gato es un carnívoro. Este conocimiento nos prohibe ponerle tierra y regarlo todas las mañanas ya que un gato no es un miembro de una clase llamada *plantas*. En cambio sabemos que la carne es una parte esencial en la alimentación de los *carnívoros*.

Un carnívoro pertenece a una clase que engloba a otras clases y que se llama *mamíferos*. De tal manera que si no encontramos un ratón que ofrecer a Manchitas, pues le podemos ofrecer un buen plato de leche. Las características de cada clase se transfieren a las clases inferiores a través de un proceso conocido como *herencia*.



Las caracteriscas de las clases superiores son heredables

En la POO la herencia asocia a cada clase derivada, las conducta y los datos asociados con la clase que hereda. Una subclase puede tener todas las características de una superclase heredadora.

Los mamíferos son animales terrestres. ¡ Bueno !, casi todos los mamíferos son animales terrestres. La ballena y el delfín son ejemplo de instancias de clase que no cumplen con esta característica. Cuando una instancia de clase se diferencia del resto de los elementos por alguna característica menor, es importante notar que esto no la excluye de la clase. Por otra parte, una instancia puede heredar conductas de más de una clase, para tener una conducta acorde con las propiedades cada una de dichas clases.

Cuando una clase hereda conductas de otra clase, el código asociado no tiene que reescribirse. Esto ahorra a los programadores muchas horas en la codificación de sistemas que reutilizan código a través de la herencia. Cuando se está trabajando en proyectos grandes se pueden formar equipos de trabajo en base a las clases emparentadas por la herencia.

La herencia acelera la rapidez con la cual se realizan los prototipos de las funciones y se asegura que

las interfaces serán consistentes en todas las clases que la utilicen. Esto evita problemas con la compatibilidad entre objetos similares. Si un error es encontrado en una clase base, este se corrige en un solo punto y se refleja en todas las clases herederas. De la misma manera, si en una clase base se reimplanta un método para que este sea más eficiente, el cambio se verá reflejado en cada clase derivada.

A decir verdad no todo es miel sobre hojuelas. La herencia es una herramienta muy valiosa cuando se trata de modelar problemas complejos. Pero añade una gran complejidad a los programas que la utilizan. El tiempo de ejecución puede llegar a incrementarse ligeramente ya que los métodos a través de los cuales se implanta la herencia ocupan mayor tiempo de procesamiento que una simple llamada. Sin embargo, el tiempo de desarrollo también disminuye cuando se utiliza esta herramienta.

IMPLANTACIÓN DE LA HERENCIA: La clase base

Una de las clases más utilizadas en los ambientes gráficos que tan de moda se han puesto en los últimos años son, las ventanas para el despliege de información y constituyen una entidad fundamental de los sistemas de software.

Escribiremos primero una sencilla clase para el despliege de una ventana en la pantalla. Suponemos que su implantación puede utilizar una función que permita colocar el cursor en una posición determinada. En caso de que no conozca la manera de realizarlo puede hacer lo siguiente. Si su sistema se encuentra basado en el PC puede utilizar una función de interrupcion al microprocesador (10H video) para escribir dicha función y hacerla "rápida". O bién, si su monitor es compatible con ANSI puede utilizar las secuencias de escape que se listan en el apendice B de estos apuntes. En los PC incluya el manejador ANSI.SYS dentro del config.sys. (device = ANSI:SYS). Vease el apendice C de estos apuntes

La clase `ventana` puede crear una ventana entre las coordenadas definidas por la esquina superior izquierda y la esquina inferior derecha. Cuando esta ventana es declarada, es decir, cuando se activa el constructor de clase solo se almacenan los valores de las coordenadas en donde será desplegada la ventana. La ventana conoce su lugar en la pantalla. A través de la función miembro `dibujar()` la ventana es capaz de dibujarse a sí misma en la pantalla. La función miembro `coloca()` establece la ventana en una posición distinta dentro de la pantalla.

El archivo de cabecera de la clase ventana es mostrado en el listado III.2.1

```
//ventana.h
// Listado III.2.1
// Este archivo solo contiene la declaracion de clase para las ventanas
// 4/abr/93
// Ernesto Peñaloza Romero



//          La declaracion de una ventana requiere de cordenada de la
// esquina superior izquierda y de la coordenada inferior derecha de
// la ventana. Cada coordenada se escribe (columna, renglon)
//
//    Una declaracion sin parametros crea un cuadro en (1,1)

#ifndef _VENTANAS_H
      #define _VENTANAS_H 1
```

```
    class ventana
    {
    public:
          ventana(int supizqy,int supizqx, int infdery, int infderx );
          ventana();
          void dibujar(void);
          void coloca(int supizqy,int supizqx, int infdery, int infderx );

    protected:
          int x1;
          int y1;
          int x2;
          int y2;
    };
#endif
```

En este archivo de cabecera se observa el uso de directivas para el preprocesador. La clase ventana es un caballito de batalla que prácticamente cualquier programa que requiera desplegar información en la pantalla utilizará. Es lógico suponer que la directiva **#include ventana.h** aparecerá con bastante frecuencia. El compilador, sin embrago, solo aceptará que se defina a la clase una sola vez. Si el compilador encuentra una variable o una clase definida en forma múltiple se confunde y marca un error fatal.

Con la directiva `#ifndef` se pregunta por la macro `_VENTANA_H` que es definida dentro de la clase. Si la macro no se encuentra definida (tal como sucede la primera vez que el compilador abre el archivo), entonces se procede a la definición de la clase y la definición de la macro. Cuando el compilador vuelva a encontrar una directiva hacia `ventana.h`, la macro ya se encontrará definida y no se intentará un redefinición de la clase.

Observese que la clase posee dos constructores. Uno que acepta parámetros enteros y uno sin parámetros. En general, es posible definir tantas funciones con el mismo nombre como sean necesarias. La única restricción es que las listas de parámetros sean escencialmente diferentes unas de otras. Es decir, no debe haber dos listas de parámetros iguales.

Para la clase **ventanas**, el constructor sin parámetros será llamado cuando se encuentre una definición con el nombre de la instancia como unico dato. Si además del nombre de la instancia, se suministran las coordenadas en donde esta se encuentra localizada, entonces se hará uso del constructor alterno que contempla esta opción. Esto nos permitirá definir la ventana de más de una manera, dependiendo de las necesidades que se tengan.

Las cordenadas de la ventana están etiquetados como **protected** es decir, estos datos solo pueden ser accesados por las funciones miembro de la clase o por clases derivadas, tal y como se verá más adelante.

Los métodos para la clase ventana se encuentran definidos en el listado III.2.2

```
// Listado III.2.2
// Implantación de los métodos de la clase ventanas
// 4/abr/93
// Ernesto Peñaloza Romero

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

```
#define ESQ_SUP_IZQ   (char)0x0DA
#define ESQ_SUP_DER   (char)0x0BF
#define ESQ_INF_IZQ   (char)0x0C0
#define ESQ_INF_DER   (char)0x0D9
#define BARRA_HORIZ   (char)0x0C4
#define BARRA_VERT    (char)0x0B3


// Constructor sin argumentos
ventana::ventana()
{
 x1=0;
 x2=0;
 y1=0;
 y2=0;
}

// Constructor con las coordenadas de la ventana
// Observese que la validación no es exahustiva
ventana::ventana(int supizqy,int supizqx, int infdery, int infderx)
{
      x1 =supizqx;
      y1= supizqy;
      if (infderx > x1)
            x2 = infderx;
      else
            x2 = x1 + 1;
      if (infdery > y1)
            y2 = infdery;
      else
            y2 = y1 + 1;

}

// Coloca las coordenadas de la ventana en una nueva posición
void ventana::coloca(int supizqy,int supizqx, int infdery, int infderx)
{
      x1 =supizqx;
      y1= supizqy;
      if (infderx > x1)
            x2 = infderx;
      else
            x2 = x1 + 1;
      if (infdery > y1)
            y2 = infdery;
      else
            y2 = y1 + 1;

}
```

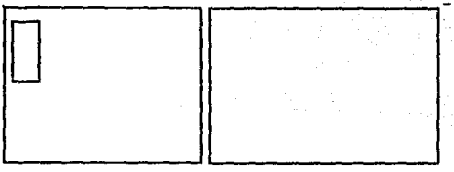
```
// Dibuja la ventana en la pantalla
void ventana::dibujar(void)
{
      register int contador;
      int longitud;


      // Esquina superior izquierda
      gotoxy(y1,x1);
      cout << ESQ_SUP_IZQ;

      // Barra horizontal superior
      longitud = y2-y1;
      if (longitud < 1)
            longitud = 1;
      for (contador=1; contador < longitud; contador ++)
      {
            cout << BARRA_HORIZ;
      }

      //Esquina superior derecha
      gotoxy(y2,x1);
      cout << ESQ_SUP_DER;

      //Barras laterales
      longitud = x2-x1;
      if (longitud <1)
            longitud = 1;
      for (contador=1; contador < longitud; contador++)
      {
            gotoxy(y1,contador+x1);
            cout<< BARRA_VERT;
            gotoxy(y2,contador+x1);
            cout << BARRA_VERT;
      }

      //Esquina inferior izquierda
      gotoxy(y1,x2);
      cout << ESQ_INF_IZQ;

      // Barra horizontal inferior
      longitud = y2-y1;
      if (longitud < 1)
            longitud = 1;
      for (contador=1; contador <longitud; contador ++)
      {
            cout << BARRA_HORIZ;
      }

      // Esquina inferior derecha
      gotoxy(y2,x2);
      cout << ESQ_INF_DER;
}
```

Los métodos listados en esta sección son bastante sencillos y no requieren de mayor explicación.

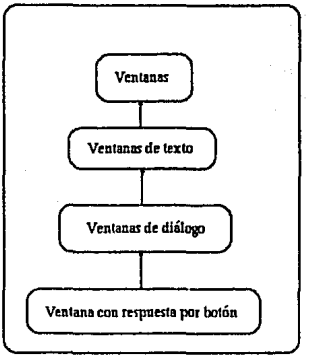
El programa principal se reduce a unas cuantas lineas que utilizan la clase ventanas. Observese el listado III.2.3.

```
// Listado III.2.3
// Utilizacion de la clase ventanas
// venmain.cpp

#include "ventana.h"
#include "ventana.cpp"
#include <conio.h>
void main(void)
{

      clrscr();
      ventana cuadrito(2,2,5,5);
      ventana marco(1,1,29,10);

      cuadrito.dibujar();
      marco.dibujar();
      marco.coloca(30,1,60,10);
      marco.dibujar();
}
```

Resultado:



En los sistemas de despliege gráfico encontramos diversos tipos de ventanas. Las ventanas de mensaje poseen solo un texto asociado a cada ventana, algunas más refinadas aceptan una respuesta, o inclusive poseen botones de radio para recibir la respuesta. Podemos entonces establecer una jerarquía de clases. Cada nueva clase tiene las características de la clase anterior más un refinamiento.



Cada clase hereda las características de su clase base

Ahora utilizaremos la clase ventana para construir una nueva clase que heredará las características de la clase base. En nuestro caso, escribiremos una clase para manejar ventanas con un texto dentro de ellas.

IMPLANTACION DE LA HERENCIA: La clase derivada

Dado que vamos a heredar de la clase ventanas esta clase no es necesaria modificarla para implantar la nueva clase. La nueva clase requiere del nombre de la clase de la cual heredará. Dicha clase recibe el nombre de *clase base*. Esta se especifica despues del nombre de la nueva clase y separada por dos puntos. La sintaxis es la siguiente:

class nombre_clase_derivada : tipo_acceso nombre_clase_base

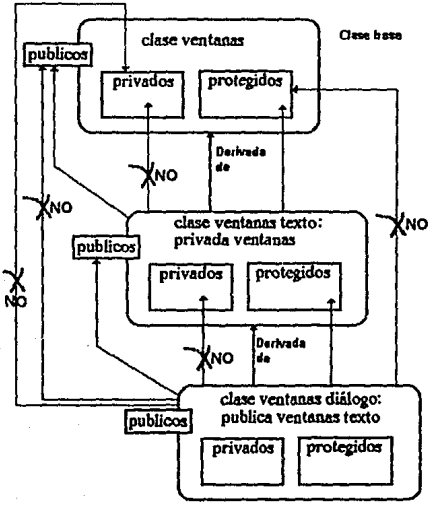
Donde:

nombre_clase_derivada:	Es el nombre de la nueva clase
tipo_acceso:	Son las palabras **public** o **private**. Por omisión el acceso es **private**.
nombre_clase_base:	Es el nombre de la clase de la cual se heredarán las características.

Puede observarse que la palabra clase y la especificación del nombre de la nueva clase son idénticas a la declaración de una clase normal. De hecho en seguida del nombre de la clase derivada se especifican los parámetros de la nueva clase. Enseguida de ello se colocan los dos puntos que determinan en forma sintáctica la herencia.

El tipo de acceso a la clase base determina como se podrán accesar los miembros de la clase base. Cuando se especifica un acceso público todos los miembros públicos y protegidos de la clase base serán públicos y protegidos para la clase derivada.

Cuando se omite el tipo de acceso este es considerado como privado. Cuando una clase base se declara como privada, todos los miembros públicos y privados de la clase base se convierten en miembros privados de la clase derivada.



Acceso a los miembros de la clase base por las clases derivadas

El listado III.2.4 muestra el archivo de encabezado de la clase que se encuentra heredando de la clase ventana. La nueva clase es llamada `aviso` y define a una ventana que puede desplegar un letrero de texto.

```
// Aviso.h
// Definicion de la clase: aviso
// Esta clase es una ventana con un mensaje de texto
// 5/abr/93
// Ernesto Peñaloza Romero
```

```
#ifndef _AVISO_H
      #define _AVISO_H

      #include "ventana.h"
      #include "ventana.cpp"
      class aviso: public ventana
      {
      public:
            char *mensaje;

            aviso(char *mssg, int x1, int y1, int x2, int y2);
            void pinta(void);
            void pinta(char *mensaje);
      };
#endif
```

El encabezado de la nueva clase se define como derivada en forma pública de la clase base. El único dato agregado es un puntero al mensaje. Las instancias de esta clase poseen las coordenadas de las dos esquinas que definen a la ventana. El constructor de la clase aviso exige dichos datos. La nueva clase posee una función miembro con dos listas de parámetros distintos.

Las funciones públicas y protegidas de la clase base son accesibles a una instancia de la clase derivada. Los métodos no necesitan ser reescritos para tener acceso a ellos, de hecho, un programador solo necesita conocer el archivo de encabezado de la clase para poder utilizarla. Solo en el caso de que sea vital conocer la manera como la clase implanta los métodos, un programador requerirá de la lectura de los programas que implantan dichos métodos.

Observese que aún no se define la forma como se comunicará la clase derivada con la clase base. La definición se realiza en la implantación de los métodos de la clase derivada. Esto puede observarse en el listado III.2.5

```
// Listado III.2.5
// aviso.cpp
// Implantacion de  los metodos de la clase aviso
// 5/abr/93
// Ernesto Peñaloza Romero

aviso::aviso(char *mssg, int y1, int x1, int y2,int  x2):ventana(y1,x1,y2,x2)
{
            mensaje = mssg;

}
void aviso::pinta(void)
{
            gotoxy (y1+2,x1+2);
            cout << mensaje;
            dibujar();
}
```

```
void aviso::pinta(char *mensaje)
{
            gotoxy (y1+2,x1+2);
            cout << mensaje;
            dibujar();
}
```

El constructor de la clase aviso define en su lista de parámetros el mensaje que será desplegado y las coordenadas de las esquinas superior izquierda e inferior derecha de la ventana. Observese la forma utilizada para especificar la herencia del contructor:

nombre_clase_derivada::nombre_clase_derivada (argumentos_1) : nombre_clase_base (argumentos_2)

Donde:

argumentos_1: Es la lista de argumentos de la clase derivada
argumentos_2: Es la lista de argumentos de la clase base. Puede haber comunicación entre la lista de *argumentos_1* y *argumentos_2*

El operador de resolución de ámbito (::) elimina cualquier posible confusión en el alcance del constructor. La lista de `argumentos_1` requiere de la definición de los tipos. Es muy importante observar que los argumentos que pasan de la lista de `argumentos_1` a la lista de `argumentos_2` no tienen definición de tipo en la segunda lista. Si se intenta definir el tipo en la segunda lista, el compilador asumirá que se intenta declarar una nueva variable y eso provocará un error

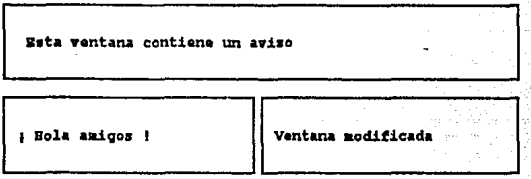
Los métodos para pintar la ventana de texto pueden ser llamados sin argumentos (el mensaje fué definido dentro del constructor) o con el mensaje como argumento (las coordenadas de la ventana no variarán entre mensaje y mensaje).

Veamos ahora el programa principal en el listado III.2.6

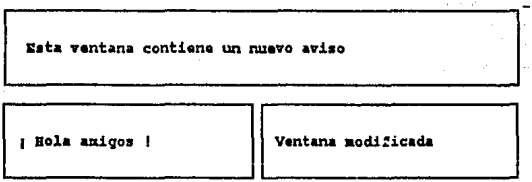
```
// Listado III.2.6
#include <conio.h>        //Prototipo de clrscr()
#include "aviso.h"
#include "aviso.cpp"
void main(void)
{
      // Crea dos instancias de aviso
      aviso principal(" ",1,1,59,5);
      aviso secundario("¡ Hola amigos !",1,6,29,10);

      clrscr();
      // En base a la ventana ya definida se despliega un mensaje
      principal.pinta(" Esta ventana contiene un aviso");

      // Se despliega la ventana de aviso en forma. La ventana está
      //completamente definida
      secundario.pinta();

      // Se utiliza la herencia de la clase base para modificar
      // los atributos del objeto y redesplegarlo
      secundario.coloca(30,6,59,10);
      secundario.pinta("Ventana modificada");
```

```
        getche();

        // Se despliega una ventana modificando solo un atributo
        principal.pinta(" Esta ventana contiene un nuevo aviso");

}
```

Resultados:

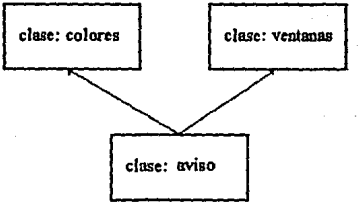


<cualquier tecla>



HERENCIA MULTIPLE

Una clase no solo puede heredar de una sola superclase. En el ejemplo expuesto con anterioridad, la clase aviso podría heredar de una superclase que establesca los colores de la pantalla. La clase **colores** puede así heredar sus conductas a la clase aviso para que esta despliege ventanas de diversos colores y con distintos atributos. La herencia de la clase aviso vendría definida en forma multiple.



La clase aviso hereda en forma multiple

La herencia múltiple no siempre es necesaria, y pocos lenguajes orientados a objetos la soportan como lo hace C++. Su sintaxis es muy sencilla. En la declaración de clase se especifican las clases de las cuales heredará la nueva clase derivada. Cada clase especifica en forma individual su tipo de acceso y separada por comas de las otras clases de las cuales hereda la clase derivada.

```
class clase_base1
{
// Definicion de la primer clase base
}
class clase_base2
{
// Definicion de la segunda clase base
}
class nombre_derivada:tipo_acceso clase_base1, tipo_acceso clase_base2
{
// definición de la clase derivada
}
```

El constructor de la clase derivada se especifica de la misma forma:

```
nombre_derivada::nombre_derivada(args_1):clase_base1(args_2),clase_base2(args2)
```

Para el ejemplo sugerido podemos definir la clase `colores` de la siguiente manera:

```
clase colores
{
      public:
            colores();
            void rojo(void);
            void verde(void);
            void azul(void);
            void normal();
            void inverso();
            ~colores(){ normal()};
}
```

La clase derivada aviso heredera en forma múltiple quedaría de la siguiente manera:

```
// Herencia múltiple
// Definicion de la clase : aviso
// Esta clase es una ventana con un mensaje de texto
// Ernesto Peñaloza Romero

#ifndef _AVISO_H
      #define _AVISO_H

      #include "colores.h"
      #include "ventana.h"
      #include "ventana.cpp"
      class aviso: public ventana, public colores
      {
      public:
            char *mensaje;

            aviso(char *msmg, int x1, int y1, int x2, int y2);
            void pinta(void);
            void pinta(char *mensaje);
      };
#endif

// Implantacion del constructor de la clase aviso

aviso::aviso(char *msmg, int y1, int x1, int y2,int  x2):
      ventana(y1,x1,y2,x2),colores()
{
            mensaje = msmg;

}
```

III.3 Métodos estáticos y métodos virtuales

ENLACE ESTÁTICO

Wirth afirma *:El tipo de resultado que una función entrega o recibe, puede ser conocido desde tiempo de compilación.* Los lenguajes diseñados bajo esta premisa suponen que una función siempre tiene que recibir un tipo de dato específico conocido antes de que el programa se encuentre en funcionamiento. A su vez el tipo al cual pertenecerá el resultado es conocido de antemano.

Las promociones de tipo y las reglas para la combinación de los tipos predefinidos de un lenguaje son la muestra más clara de este tipo de pensamiento. Si en una expresión matemática se tiene:

resultado = real operacion entero

El resultado será de tipo real, ya que , independientemente del tipo de operación que se trate, existe un elemento real en la expresión que evita un resultado entero. Recuerdese que aquellas operaciones que arrojan resultados enteros, involucran numeros enteros a ambos lados del operador.

Esta filosofía de pensamiento implica un total conocimiento de los tipos de datos que se requieren y de los resultados arrojados. Ejemplifiquemos esto de la siguiente manera.

En un sistema de Diseño Asistido por Computadora podríamos tener diversas figuras simétricas básicas. Para generar cada una de las figuras realizamos una llamada auxiliandonos de un menú. Vease el listado III.3.1

```
// Listado III.3.1
#include <iostream.h>
#include <conio.h>

void pidecirculo(void)
{
      cout << " circulo";
}
void pidecuadro(void)
{
      cout <<" cuadro";
}
void pidetriangulo(void)
{
      cout <<" triangulo";
}

void main(void)
{
      int opcion;

      clrscr();
      cout << "1.- Circulo"<<endl;
      cout << "2.- Cuadro "<<endl;
      cout << "3.- Triangulo"<<endl;
      cout <<endl<<"Opcion: " ;

      do
      {
            cin >>opcion;
      }while (opcion <1 || opcion >3);

      cout << endl <<"\t Pidiendo datos de: ";
```

```
        switch (opcion)
        {
                case 1:     pidecirculo();
                            break;

                case 2:     pidecuadro();
                            break;

                case 3:     pidetriangulo();
                            break;
        }
}
```

Tal y como puede observarse, las llamadas a todas y cada una de las funciones se encuentra predeterminada en base a la opción del menú que el usuario elija. El ligado de las direcciones en donde se encuentran las funciones se realiza en forma única durante el tiempo de compilación.

Durante el proceso de compilación, el código fuente es traducido a código objeto. A cualquier función le es asignada una dirección única. Dependiendo del sistema operativo, esta dirección puede ser absoluta o relativa al punto de inicio. Independientemente de ello, la función poseerá un punto de entrada único

Esto es llamado *enlace estático* porque se realiza una sola vez en todo el proceso. Una función con enlace estático es conocida como función estática. En inglés se conoce al anlace estático como *static binding.*

LLAMADAS A FUNCIONES USANDO PUNTEROS

Hemos utilizado los punteros para direccionar a variables o estructuras de datos almacenados en la memoria. Si hacemos más extensiva esta idea encontraremos que una función es una entidad que posee una dirección (su punto de entrada) y por tanto podemos manejar un puntero a una función. La forma general es la siguiente:

*tipo_devuelto (*nombre_puntero_funcion)()*

Ahora implantaremos el mismo ejemplo de la sección anterior con punteros a funciones:

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

void pidecirculo(void)
{
        cout << " circulo";
}
void pidecuadro(void)
{
        cout <<" cuadro";
}
void pidetriangulo(void)
{
        cout <<" triangulo";
}

void main(void)
{
        int opcion;
```

```
void (*tablaordenes[1]) () =
{
      pidecirculo,
      pidecuadro,
      pidetriangulo
};


clrscr();
cout << "1.- Circulo"<<endl;
cout << "2.- Cuadro "<<endl;
cout << "3.- Triangulo"<<endl;
cout <<endl<<"Opcion: " ;

do
{
      cin >>opcion;
}while (opcion <1 || opcion >3);

cout << endl <<"\t Pidiendo datos de: ";

// Recuerde que los arreglos inician en cero
opcion--;
(*tablaordenes[opcion]) ();


}
```

Como puede observarse hemos creado un arreglo de punteros a funciones. En el arreglo se han escrito los nombres de las funciones que serán tratadas en forma especial por el compilador para obtener sus direcciones y asignarlas en *tiempo de ejecucion* a la última linea de código que es donde se realiza la llamada a la función.

La función que será llamada es totalmente desconocida en tiempo de compilación. El enlace no puede realizarse antes de que el programa se ejecute sino hasta el momento en que el usuario ha decidido quer función será la que reciba la llamada . Es claro que el usuario no es conciente de todo esto ya que el proceso es transparente para él.

El enlace con la función ha sido pospuesto hasta el tiempo de ejecución, por lo que este tipo de enlace recibe el nombre inglés de *late binding (enlace tardío)* o *dynamic binding (enlace dinámico)*. Una función con enlace dinámico es conocida como función dinámica o virtual.

El término virtual se refiere a su estado de aparente existencia dentro del código. En realidad ninguna función virtual se encuentra realmente enlazada en el código hasta el momento en que esta es ejecutada y ligada en tiempo de ejecución.

El enlace dinámico es de vital importancia para poder implantar el polimorfismo.

III.4 Polimorfismo

En más de un caso los objetos pueden responder de muy diferente manera a un mismo mensaje. Por ejemplo se podría pensar en un sistema que puede comunicarse con un usuario a través de un monitor de video, o a través de un teletipo, de una terminal punto de venta, o una impresora. La funcion miembro **`Desplegar()`** de un objeto podría enviar un mensaje como:

objeto.Deplegar("En linea")

Las operaciones necesarias para resolver esta mensaje dependen totalmente del dispositivo de salida. En unos habrá que limpiar la pantalla, mientras que otros tendrán que dar un salto de carro o simplemente enceder un foquito.

A la capacidad que tiene una clase de responder de manera diferente a un mismo mensaje se le conoce como *Polimorfismo*.

Normalmente los lenguajes restringen el paso de parámetros a un solo tipo de dato. La función prototipo del lenguaje C es una clara muestra de ello. Sin embargo los lenguajes que soportan el polimorfismo permiten que una función acepten distintos tipos de datos. El signo de suma (+) es un claro ejemplo de una función polimorfica. Lo mismo puede operar sobre numeros reales, enteros con signo o sin él.

En C++ el polimorfismo sobre operadores recibe el nombre de *sobrecarga* . La sobrecarga de operadores se explicará ampliamente en el epígrafe III.7.

El polimorfismo nos permite implantar clases que respondan a un solo mensaje. Las características y los métodos con los cuales se implemente el mensaje es algo, que en general, no le interesa al objeto que realiza la llamada.

CLASES ABSTRACTAS

En este momento podemos afirmar que una clase puede estar constituida por un objeto, pero un objeto no necesariamente es una clase. La siguiente pregunta es: ¿ Puede una clase no tener instancias?

La respuesta es sí. Pueden existir clases que no creen instancias de sí mismas. Estas clases son de un tipo tal que sirven de base para aquellas que hereden sus características. Pero que no tienen utilidad alguna si se crean instancias de ellas.

Imaginemos una clase llamada **`Magnitud`**. Esta clase estaría destinada a comparar dos objetos con el fin de determinar cual de ellos es el mayor. Es bastante claro que los métodos para implantar dicha clase no pueden ser útiles si se les restringe a un cierto tipo de objeto. No es lo mismo comparar numeros reales, numeros complejos , cadenas de caracteres, matrices, etc. La clase **`Magnitud`**, para ser realmente útil, debe abstraer las características escenciales de una comparación y heredarlas a las clases derivadas .

Una clase sin instancias y que sirve de cimiento para la creación de clases derivadas es conocida como *clase abstracta*. Una clase abstracta puede ser implantada de dos maneras diametralmente opuestas. Por un lado se pueden usar como plataformas de comunicación entre las clases derivadas para que todas ellas se encuentren relacionadas. O bien suministrando para todas ellas los métodos requeridos para su funcionamiento. En este último caso, si una clase derivada requiere de un método particular para la implantación puede redefinir el método heredado y utilizar el propio.

POLIMORFISMO EN C++

En el epígrafe anterior se detalló la manera en la cual un compilador de C puede manejar funciones con enlace tardío. Como suele ocurrir con la mayoría de las cosas en C++, el enlace dinámico se ejecuta tras bambalinas. El programador solo tiene que definir a un método como virtual y el compilador se encargará de lo demás.

Una función virtual solo requiere de la antecesión de la palabra reservada `virtual` para indicar al compilador que deberá ser tratada en forma especial.

Para el ejemplo previo simplemente se tendría que definir una clase que manejara a cada uno de los comandos de lectura:

```
class orden
{
      public:
      virtual void pidedatos(void) const;
}
```

Cada objeto tendría entonces una función miembro con la misma función prototipo:

```
class circulo: public orden
{
// constructores, destructores y otras funciones

      virtual void pidedatos(void) const;
}
class cuadro: public orden
{
// constructores, destructores y otras funciones

      virtual void pidedatos(void) const;
}
cass triangulo: public orden
{
// constructores, destructores y otras funciones

      virtual void pidedatos(void) const;
}
```

Puede observarse que cada función posee un miembro que tiene exactamente el mismo nombre. Son las clases derivadas las encargadas de implantar los respectivos métodos.

FUNCIONES VIRTUALES PURAS

Si una de las clases derivadas no implanta en forma explícita un método, la clase abstracta puede emitir un mensaje de advertencia:

```
virtual void pidedatos(void)
{
      cout << "La clase derivada no ha implantado este metodo",endl;
}
```

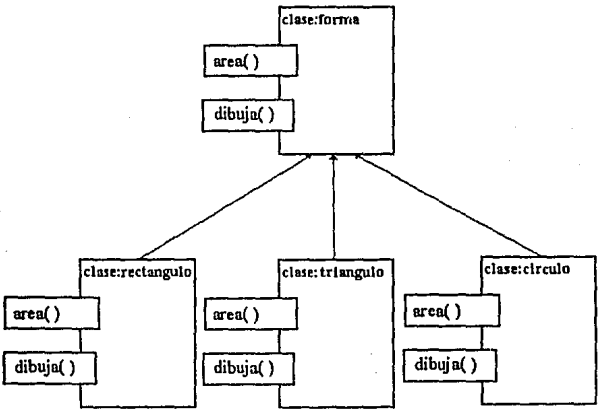
Aunque esta no es una mala idea, C++ provee las funciones virtuales puras. Con ellas el compilador detecta el error en tiempo de compilación. De esta manera se puede detectar el problema en la computadora de desarrollo y no en la máquina del usuario. Situación mil veces molesta, tanto para el usuario como para el programador. La sintaxis es muy sencilla, para el ejemplo mostrado hasta este momento la función se declararía como virtualmente pura de la siguiente manera:

```
class orden
{
      public:
      // Función virtualmente pura
      virtual void pidedatos(void) = 0;
}
```

El compilador no permitira la creación de instancias conteniendo funciones virtualmente puras. Esto es exactamente lo que queremos ya que la clase abstracta es solo un manejador de clases derivadas. Por otra parte, el compilador checa que las clases derivadas implantarán los métodos de una fucnión virtualmente pura. Si una clase derivada inmediata no puede establecer un método, esta simplemente pasa el problema a una de las clases derivadas y es declarada como virtualmente pura.

Ejemplo:

Escribiremos un ejemplo que ilustre los conceptos vertidos en este epígrafe. En un sistema se pueden manejar diversas figuras: círculos, triángulos y rectángulos. El cálculo de las áreas de estas figuras requiere de distintos métodos. Sin embargo el concepto es el mismo en todas ellas: Extraer el área de una figura.



Polimorfismo:Varias clases derivadas poseen distintos métodos para una misma idea

El listado III.4.1 es el archivo de cabecera de las clases involucradas en este ejemplo: Cada figura puede obtener su área y dibujarse a sí misma. Para definir cada figura usamos sus coordenadas dentro de la pantalla.

```
// Listado III.4.1
// FORMAS.H
// Archivo  de encabezado para las diversas formas que puede
// tener una forma geometrica
// Ejemplo de polimorfismo, clase abstracta y funciones virtuales
// 6/abr/93
// Ernesto Peñaloza Romero

#ifndef _FORMA_H
      #define _FORMA_H

      // Esta es una clase abstracta. Ninguno de sus metodos
      // se implanta en forma explicita
      class forma
      {
            public:
                  // Estos metodos son puramente virtuales
                  virtual double area(void) const=0;
                  virtual void dibuja(void) const=0;
      };

      // Clase circulo: Derivada publicamente de la clase forma
      //
      // Emplea la coordenada del centro y el radio
      // para definir una instancia de clase
      // Ejemplo: Un circulo con centro en (4,5) y radio de 3
      //          Circulo rueda(4,5,3)

      class circulo: public forma
      {
            public:
            circulo(double,double,double);
            virtual double area(void) const;
            virtual void dibuja(void) const;

            private:
                  double xc,yc; // Coordenadas del centro
                  double radio;
      };

      // Clase rectangulo: Derivada de forma
      // Emplea las coordenadas de la esquina superior izquierda
      // y la esquina inferior derecha para definir al rectangulo
      // Ejemplo: Un rectangulo con coordenadas en (1,1) y (5,5)
      //          rectangulo cuadro(1,1,5,5)
```

```
class rectangulo: public forma
{
      public:
            rectangulo(double , double , double , double );
            virtual double area(void) const;
            virtual void dibuja(void) const;

      private:
            double base;
            double altura;
            double supizqx; //Cordenada superior izquierda
            double supizqy;
            double infderx; //Coordenada inferior izquierda
            double infdery;
};

// Clase triangulo: Derivada de la clase forma
// Emplea las coordenadas de los tres vertices para su definicion
// Ejemplo: Coordenadas (1,1), (5,5) (3,3)
//     triangulo tri(1,1,5,5,3,3)


class triangulo: public forma
{
      public:
            triangulo(double,double,double,double,double,double) ;
            virtual double area(void) const;
            virtual void dibuja(void) const;

      private:
            double vertx1;
            double vertx2;
            double vertx3;
            double verty1;
            double verty2;
            double verty3;
};
#endif
```

Se observa que la clase **forma** es una clase abstracta y que todos sus métodos son puramente virtuales. Cada una de las clases derivadas utiliza la misma función prototipo para relacionarse con la clase base.

Estudiemos ahora el listado III.4.2 que contiene los métodos de las clases

```
#include <math.h> // para la definicion de M_PI
#include <conio.h>
#include <iostream.h>
// Metodos para la clase circulo
```

```
// Constructor
circulo::circulo(double centrox=0, double centroy=0, double radius=0)
{
      xc=centrox;
      yc=centroy;
      radio=radius;
}

double circulo::area(void) const
{
      return( M_PI * radio* radio);
}
void circulo::dibuja(void) const
{
      cout << endl << "Circulo: \n\tCentro: ("<< xc <<","<<yc<<")";
      cout << "\n\tRadio:"<<radio;
}

// Metodos para la clase rectangulo
// Constructor
rectangulo::rectangulo(double x1=0, double y1=0, double x2=0, double y2 =0)
{
      base=x2-x1;
      altura=y2-y1;
      supizqx=x1;
      supizqy=y1;
      infderx=x2;
      infdery=y2;
}
double rectangulo::area(void) const
{
      return (base*altura);
}
void rectangulo::dibuja(void) const
{
      cout << endl << "Rectangulo:";
      cout << endl << "\tBase   : "<< base <<endl << "\tAltura: "<<altura;
      cout << endl << "\tCoordenadas:";
      cout << endl << "\t\tSuperior izquierda: (" << supizqx<<","<<supizqy<<")";
      cout << endl << "\t\tInferior derecha  : (" << infderx<<","<<infdery<<")";
}


//Metodos para la clase triangulo
//Constructor
triangulo::triangulo(double x1=0, double y1=0, double x2=0, double y2=0,double
x3=0, double y3=0)
{
            vertx1=x1;
            vertx2=x2;
            vertx3=x3;
```

```
            verty1=y1;
            verty2=y2;
            verty3=y3;
}

double triangulo::area(void) const
{
      double x21,y21,x31,y31;
      x21= vertx2 - vertx1;
      y21= verty2 - verty1;
      x31= vertx3 - vertx1;
      y31= verty3 - verty1;
      return( fabs(y21*x31-x21*y31) /2.0 );
}

void triangulo::dibuja(void) const
{
      cout << endl << "Triangulo:";
      cout << endl << "\tCoordenadas:";
      cout << endl << "\t\tVertice 1: ("<< vertx1<<","<<verty1<<")";
      cout << endl << "\t\tVertice 2: ("<< vertx2<<","<<verty2<<")";
      cout << endl << "\t\tVertice 3: ("<< vertx3<<","<<verty2<<")";
}
```

En forma explícita cada clase implanta sus métodos para el cálculo del area de cada figura. Finalmente veamos como se utiliza. El listado III.4.3 muestra el listado del programa `main()` que utiliza las clases definidas con anterioridad.

```
// Polimorfismo.
// Este programa crea tres "instancias" de una clase abstracta
// Ernesto Peñaloza Romero
// 5/abr/93

#include "iostream.h"
#include "formas.h"
#include "formas.cpp"
#define num_fig 3

void main(void)
{
      // Punteros de la clase abstracta
      forma *figura[num_fig];

      int i;

      // Instancias de las clases concretas
      circulo rueda(5,5,2);
      rectangulo cuadro(1,1,5,5);
      triangulo tri(1,1,1,5,5,5);

      // Asignacion a la clase abstracta
      figura[0] = &rueda;
      figura[1] = &cuadro;
      figura[2] = &tri;
```

```
        // Se despliegan los datos de cada figura
        // Se calcula su area
        clrscr();
        for (i=0; i< num_fig ; i++)
        {
                figura[i]->dibuja();
                cout << endl << "\tArea: " << figura[i]->area()<<endl;
        }
}
```

Resultados:

```
Circulo:
        Centro: (5,5)
        Radio:2
        Area: 12.566371

Rectangulo:
        Base  : 4
        Altura: 4
        Coordenadas:
                Superior izquierda: (1,1)
                Inferior derecha  : (5,5)
        Area: 16

Triangulo:
        Coordenadas:
                Vertice 1: (1,1)
                Vertice 2: (1,5)
                Vertice 3: (5,5)
        Area: 8
```

El programa crea tres punteros a objetos de la clase abstracta. Dichos punteros conforman un arreglo de punteros que toma el nombre de `figuras`. Es por demás evidente que estos punteros deben hacer referencia a determinados objetos concretos que serán creados con posterioridad. En el ejemplo se crean tres instancias concretas: un círculo, un triángulo y un rectángulo.

Debido a que el objeto `figura[i]` es un puntero, la llamada a cualquiera de las funciones miembro debe realizarse con el operador ->. Ahora bien, un conocimiento que cualquier objeto posee y que no hemos mencionado explícitamente es su "*conciencia*" sobre la clase a la cual pertenece.

Cuando se realiza la llamada del puntero `figura[0]->dibuja()` se realiza la llamada en la clase abstracta de la forma `forma.dibuja()`. Dado que esta es una clase abstracta se inquiere por el objeto apuntado que desea accesar al método . El objeto apuntado sabe que pertenece a la clase círculo y el método que es llamado es por tanto `circulo::dibuja()`.

III.5 Objetos dinámicos

Las sentencias NEW y DELETE que se han expuesto en un apartado anterior (III.3) crean objetos en forma dinámica. En esta sección mostraremos un ejemplo completo de programación orientada a objetos.

GARABATOS.EXE

El sistema que desarrollaremos en esta sección parte del ejemplo que hemos escrito en el epígrafe anterior. Escribiremos un sistema que sea capaz de dibujar círculos, triangulos, lineas y puntos para poder realizar un dibujo sencillo. El sistema recibirá el nombre de GARABATOS

Dado que el programador no puede conocer de antemano cuantas figuras se requerirán, la asignación de cada nuevo objeto se realizará en forma dinámica. El sistema tendrá la capacidad de dibujar las figuras en la pantalla y borrar figuras de la lista de objetos que conforman el dibujo. Para hacer el programa más interesante lo escribiremos totalmente utilizando el ratón.

LA INTERFACE GRAFICA DE BORLAND

En primer instancia refinaremos el código escrito en el ejemplo precedente. Deseamos escribir un programa que *realmente* dibuje las figuras en la pantalla. Si usted no posee el compilador de Borland puede escribir sus funciones gráficas a base de interrupciones al microprocesador.

La interface grafica de Borland requeire de un conjunto de archivos con extensión .BGI. Estos archivos contienen la información para poder utilizar distintos tipos de monitores. La detección del tipo de monitor que la computadora posee se realiza en el momento de la inicialización del las rutinas graficas.

GARABATOS hace uso de una clase a la cual llamaremos pantalla. Esta clase habilitará el modo grafico, detectará el tipo de monitor existente y cargará el manejador de pantalla que sea requerido. Los prototipos y las macros necesarias de encuentran en el archivo de encabezado `graphics.h`

Veamos el listado.

```
/* GRAFLIB.CPP
   Clase que permite el acceso a las Librerias Graficas  */

#ifndef PANTA_H
      #define PANTA_H 1

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

      class pantalla
      {

            public:
                  pantalla(void)
                  {
                        Inicializa();
                  }
```

```
                ~pantalla(void)
                {
                      Cierra();
                }
                void Inicializa(void);
                void Cierra(void);
      };


      void pantalla::Inicializa(void)
      {
        int ManejaGraf,ModoGraf;
        int ErrorCod;

        // Coloca el modo de autodetección
        ManejaGraf = DETECT;
        // Inicializa los graficos
        initgraph( &ManejaGraf, &ModoGraf, "" );

        // Lee el resultado de la inicialización
        ErrorCod = graphresult();
        // Si (existio un error)
        if( ErrorCod != grOk )
        {
            /* Se detecto un error fatal durante la inicialización     */
          printf(" Error en el Sistema Gráfico: %s\n", grapherrormsg( ErrorCod ));
          exit( 1 );
        }
      }
      void pantalla::Cierra(void)
      {
            closegraph();
      }
#endif
```

Observaciones:

Esta clase nos servirá para activar todas las funciones graficas que GARABATOS requiere. La clase solo posee una función miembro con funciones totalmente nuevas. La función miembro `Inicializa` declara las variables `ManejaGraf` (que determinará el modo de operación de la función `initgraph()`) y `ModoGraf` que devolverá el modo grafico con el cual se ha inicializado el monitor. La primera sentencia ejecutable coloca el modo de inicialización en detección automática. La inicialización de los graficos se realiza con la función `initgraph()` que es parte de las funciones de Borland. El tercer parámetro requerido por la función es la trayectoria en donde se encuentran los archivos BGI. En nuestro ejemplo los supondremos instalados en el directorio en donde se coloque el programa ejecutable.

Dado que nada asegura que la inicialización haya sido satisfactoria se pregunta por el estado de la inicialización. La función `graphresult()` también forma parte de las funciones que proporciona Borland. Esta función retorna un numero entero que indica el tipo de error generado durante la inicialización de los graficos. La macro `grOk` define el valor entero que la función `graphresult()` devuelve si la inicialización gráfica resultó exitosa.

El destructor de la clase solo consta de la función **closegraph()** que se encarga de cerrar las funciones graficas.

LA CLASE RATON

La clase raton será de gran utilidad en GARABATOS. El ratón que utiliza el sistema debe ser compatible con el ratón de Microsoft y debe utilizar la interrupción 33 del microprocesador. Aunque existen diversas librerías para el manejo del ratón, se ha preferido exponer la forma como puede manejarse este dispositivo sin contar con más ayuda que el manejador proporcionado por el fabricante del ratón.

Un ratón es un dispositivo que coloca un puntero en la pantalla. Este puntero navega por la pantalla conforme el usuario mueve el dispositivo en su mesa de trabajo. El ratón suele poseer como mínimo dos botones. Estos botones reciben el nombre de su posicón; es decir, se tiene un boton derecho y un boton izquierdo. En un momento dado, el usuario puede oprimir cualquiera de estos botones e inclusive puede orpimir ambos botones a la vez.

El punotro del ratón consiste de un cuadro de color, cuando el monitor se encuentra en el modo de texto, y se muestra como una flecha cuando el monitor se encuentra en el modo grafico. Este despliege es realizado automáticamente por el manejador del dispositivo. El puntero puede aparecer y desaparecer a voluntad del programador. También es posible establecer el puntero en una posición determinada.

```
// RATON:CPP
// Clase raton
// Utiliza la interrupcion 33 del microprocesador
// Supone la existencia de un manejador para raton Microsoft
// Programo: Ernesto Penaloza Romero


#ifndef RATON_H 1

      #define RATON_H

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

      #define RATON 0x33              // Numero de la interrupcion
      #define BO_NINGUNO    0
      #define BO_DERECHO    1
      #define BO_IZQUIERDO  2
      #define BO_AMBOS      3


      class raton
      {
      public:
            raton()
            {
                  inicia_raton();
            };
```

```
        void inicia_raton(void);
        void apaga_raton(void);
        void coloca_raton(int x, int y);
        void posicion(int & x, int & y);
        void esconde_raton(void);
        void aparece(void);
        int boton(void);
};

// Activa el raton
void raton::inicia_raton( void )
{
        union REGS regs;

        regs.x.ax = 1;     // Activa el raton
        regs.x.cx = 100;
        regs.x.dx = 100;
        regs.x.bx = BO_NINGUNO;
        int86(RATON, &regs, &regs);
}

// Coloca el raton en una posici¢n determinada
void raton::coloca_raton( int x, int y)
{
        union REGS regs;

        regs.x.ax = 4;        //Establece el raton
        regs.x.cx = y;
        regs.x.dx = x;
        regs.x.bx = BO_NINGUNO;
        int86(RATON, &regs, &regs);
}

// Desactiva el raton
void raton::apaga_raton( void )
{
        union REGS regs;

        regs.x.ax = 0;  /* Desactiva el raton  */
        int86(RATON, &regs, &regs);
}

//Obtiene la posici¢n del raton
void raton::posicion( int & x, int & y )
{
        union REGS regs;

        regs.x.ax = 3;           /*  Modo de pregunta  */
        regs.x.bx = BO_NINGUNO;
        int86(RATON, &regs, &regs);
        x = regs.x.dx;     /* posici¢n del rat¢n: renglon */
        y = regs.x.cx;     /* posic¡¢n del rat¢n: columna */
}
```

```
        // Aparece el raton
        void raton::aparece( void )
        {
                union REGS regs;

                regs.x.ax = 3;          /*  Modo de pregunta  */
                regs.x.bx = BO_NINGUNO;
                int86(RATON, &regs, &regs);
        }

        // Obtiene el boton que ha sido oprimido
        int raton::boton( void )
        {
                union REGS regs;

                regs.x.ax = 3;          /*  Modo de pregunta  */
                regs.x.bx = BO_NINGUNO;
                int86(RATON, &regs, &regs);
                return( (int) regs.x.bx);
        }

        // Desaparece el indicador grafico de la pantalla, sin
        // desactivar el raton
        void raton::esconde_raton( void )
        {
                union REGS regs;

                regs.x.ax = 2;  /* Esconde el raton */
                regs.x.bx = BO_NINGUNO;
                int86(RATON, &regs, &regs);
        }
#endif
```

EL REFINAMIENTO DE LAS FIGURAS

Las clases de figuras que hemos escrito en el ejemplo de polimorfismo aún nos son de utilidad. Hemos agregado dos clases más. La clase linea, que dibujará lineas rectas y la clase punto que pintará un pixel en la pantalla. Tal y como puede observarse, los cambios son mínimos en el archivo de encabezado y solo se refieren a la inclusión de las nuevas figuras. El único cambio significativo lo constituye la nueva función miembro que checa la correcta inicialiñización de los graficos. Dado que los constructores utilizarán un ratón, se hace un llamado a la clase ratón. De momento no es necesario que sapamos como se implanta.

```
// FORMAS.H
// Archivo  de encabezado para las diversas formas que puede
// tener una forma geometrica
// Fecha de creacion:      6/abril/1993
// ultima actualizacion: 26/mayo /1993
// Programo: Ernesto Peñaloza Romero
```

```
#ifndef _FORMA_H
      #define _FORMA_H
      #include "raton.cpp"

      // Esta es una clase abstracta. Ninguno de sus metodos
      // se implanta en forma explicita
      class forma
      {
            public:
                  // Estos metodos son puramente virtuales
                  virtual double area(void) const=0;
                  virtual void dibuja(void) const=0;
                  virtual void datos(void)  const=0;
                  virtual void borra(void)  const=0;

                  int graficos_ok(void)
                  {
                    int correcto=1;
                    if (graphresult() != grOk)
                    {
                        printf("\nError Grafico\n");
                        printf("Pulse cualquier tecla");
                        getch();
                        correcto=0; /* Termina indicando un error */
                    }
                    return (correcto);
                  };
            protected:
                  raton mimi;
      };



      // Clase circulo: Derivada de publicamente de la clase forma
      //
      // Emplea la coordenada del centro y el radio
      // para definir una instancia de clase
      // Ejemplo: Un circulo con centro en (4,5) y radio de 3
      //          Circulo rueda(4,5,3)

      class circulo: public forma
      {
            public:
            circulo();
            circulo(double,double,double);

            virtual double area(void) const;
            virtual void dibuja(void) const;
            virtual void datos(void)  const;
            virtual void borra(void)  const;
```

```
      private:
            int xc,yc; // Coordenadas del centro
            int radio;
};

// Clase rectangulo: Derivada de forma
// Emplea las coordenadas de la esquina superior izquierda
// y la esquina inferior derecha para definir al rectangulo
// Ejemplo: Un rectangulo con coordenadas en (1,1) y (5,5)
//          rectangulo cuadro(1,1,5,5)


class rectangulo: public forma
{
      public:
            rectangulo();
            rectangulo(double , double , double , double );
            ~rectangulo()
               {
                     borra();
               };
            virtual double area(void) const;
            virtual void dibuja(void) const;
            virtual void datos(void) const;
            virtual void borra(void)  const;

      private:
            double base;
            double altura;
            int supizqx; //Cordenada superior izquierda
            int supizqy;
            int infderx; //Coordenada inferior izquierda
            int infdery;
};

// Clase triangulo: Derivada de la clase forma
// Emplea las coordenadas de los tres vertices para su definicion
// Ejemplo: Coordenadas (1,1), (5,5) (3,3)
//    triangulo tri(1,1,5,5,3,3)


class triangulo: public forma
{
      public:
            triangulo();
            triangulo(double,double,double,double,double,double) ;
            virtual double area(void) const;
            virtual void dibuja(void) const;
            virtual void datos(void)  const;
            virtual void borra(void)  const;
```

```
        private:
                int vertx1;
                int vertx2;
                int vertx3;
                int verty1;
                int verty2;
                int verty3;
};

// Clase linea: Derivada de la clase forma
// Emplea las coordenadas de los dos puntos que definen la linea
// Ejemplo: Coordenadas (1,1), (5,5)
//    linea raya(1,1,5,5)


class linea: public forma
{
        public:
                linea();
                linea(double,double,double,double) ;
                virtual double area(void) const;
                virtual void dibuja(void) const;
                virtual void datos(void)  const;
                virtual void borra(void)  const;

        private:
                int supizqx;
                int infderx;
                int supizqy;
                int infdery;
};

// Clase punto: Derivada de la clase forma
// Emplea la coordenada del punto
// Ejemplo: Coordenadas (1,1)
//    punto pixel(1,1)


class punto: public forma
{
        public:
                punto();
                punto(double,double) ;
                virtual double area(void) const;
                virtual void dibuja(void) const;
                virtual void datos(void)  const;
                virtual void borra(void)  const;

                void coloca(double,double);
```

```
        private:
              int x;
              int y;
    );
#endif
```

IMPLANTACION DE LAS CLASES DERIVADAS DE LA CLASE FORMA

Los cambios más significativos en las implantaciones de las formas a desplegar lo constituyen los constructores de las formas, que ahora se realizan por medio del ratón. La función miembro graficos_ok() y una instancia de la clase raton son heredadas de la clase forma.

Un raton puede hacer diversas cosas. En primer lugar, debe activarse, esconder el puntero grafico del ratón, obtener su posición en la pantalla y colocarse en una posicón especifica. Más adelante veremos la forma de implantar estos métodos.

Cada constructor puede crera una figura en base a las posición del ratón. Las coordenadas se establecen cuando el usuario pulsa el botón derecho del ratón. Veamos los métodos:

```
#ifndef FORMAS_CPP
      #define FORMAS_CPP
      #include <math.h> // para la definicion de M_PI
      #include <conio.h>
      #include <iostream.h>

      #define PREVIO LIGHTRED


      // Metodos para la clase circulo

      // Constructor
      circulo::circulo()
      {
            int color;
            int radiox, radioy;

            // Centro del circulo

            // Hasta que se oprima el boton derecho
            while (mimi.boton() != 1);
            // Obtiene posicion del cursor del raton
            mimi.posicion (xc,yc);

            color=getcolor();
            setcolor(PREVIO);
```

```
        // Radio
        while (mimi.boton() !=1);
        while (mimi.boton() == 1)
        {
              mimi.posicion (radiox,radioy);
              radio = sqrt( pow((radiox-xc),2)+pow((radioy-yc),2) );
              mimi.esconde_raton();
              dibuja();
              borra();
              mimi.inicia_raton();
        }
        mimi.posicion (radiox,radioy);
        radio = sqrt( pow((radiox-xc),2)+pow((radioy-yc),2) );
        setcolor(color);
        mimi.esconde_raton();
        dibuja();
        mimi.inicia_raton();
}
circulo::circulo(double centrox, double centroy, double radius)
{
        xc=centrox;
        yc=centroy;
        radio=radius;
}

double circulo::area(void) const
{
        return( M_PI * radio* radio);
}
void circulo::datos(void) const
{
        cout << endl << "Circulo: \n\tCentro: ("<< xc <<","<<yc<<")";
        cout << "\n\tRadio:"<<radio;


}
void circulo::dibuja(void) const
{
   /* dibuja un circulo */
   if (graficos_ok())
         circle(yc, xc, radio);
}
void circulo::borra(void) const
{
        int color;
        color = getcolor();
        setcolor(getbkcolor());
        if (graficos_ok())
              circle(yc,xc,radio);
    setcolor(color);
}
/*
```

```
void circulo::mover(int x, int y) const
{
      borra();
      xc=x;
      yc=y;
      dibuja();
}
*/


// Metodos para la clase rectangulo
// Constructor
rectangulo::rectangulo()
{
      int color;
      // Mientras no se pulse el boton derecho
      while (mimi.boton() != 1);
      mimi.posicion (supizqx,supizqy);
      color = getcolor();
            setcolor(PREVIO);
      while (mimi.boton() == 1)
      {
            mimi.posicion (infderx,infdery);
            mimi.esconde_raton();
            dibuja();
            borra();
            mimi.inicia_raton();
      }
      mimi.posicion (infderx,infdery);
      base=infderx-supizqx;
      altura=infdery-supizqy;
      setcolor(color);
      mimi.esconde_raton();
      dibuja();
      mimi.inicia_raton();
}
rectangulo::rectangulo(double x1, double y1, double x2, double y2)
{
      base=x2-x1;
      altura=y2-y1;
      supizqx=x1;
      supizqy=y1;
      infderx=x2;
      infdery=y2;
}
double rectangulo::area(void) const
{
      return (base*altura);
}
```

```
void rectangulo::datos(void) const
{
      cout << endl << "Rectangulo:";
      cout << endl << "\tBase   : "<< base <<endl << "\tAltura: "<<altura;
      cout << endl << "\tCoordenadas:";
      cout << endl << "\t\tSuperior izquierda: (" <<
supizqx<<","<<supizqy<<")";
      cout << endl << "\t\tInferior derecha  : (" <<
infderx<<","<<infdery<<")";
}

void rectangulo::dibuja( void ) const
{

      if (graficos_ok())
      /* Dibuja un rectangulo  */
      rectangle(supizqy,supizqx,infdery,infderx);
}
void rectangulo::borra(void) const
{
      int color;
      color = getcolor();
      setcolor(getbkcolor());
      if (graficos_ok())
            rectangle(supizqy,supizqx,infdery,infderx);
      setcolor(color);
}
/*
void mover(int x, int y) const
{
      borra();
      base=x2-x1;
      altura=y2-y1;
      supizqx=x;
      supizqy=y;
      infderx=x2;
      infdery=y2;
*/


//Metodos para la clase triangulo
//Constructor
triangulo::triangulo()
{
      int color;

      // Primer vertice
      while (mimi.boton() != 1);
      mimi.posicion (vertx1,verty1);
      color=getcolor();
      setcolor(PREVIO);
      putpixel(verty1,vertx1,getcolor());
```

```
            // Siguiente vertice
            while (mimi.boton() ==1)
            {
                  mimi.posicion (vertx2,verty2);
                  mimi.esconde_raton();
                  // Pinta una linea
                  line(verty1,vertx1,verty2,vertx2);
                  // Borra la linea
                  setcolor(getbkcolor());
                  line(verty1,vertx1,verty2,vertx2);
                  setcolor(PREVIO);
                  mimi.inicia_raton();
            }

            // tercer vertice
            while (mimi.boton() !=1);
            while (mimi.boton() == 1)
            {
                  mimi.posicion (vertx3,verty3);
                  mimi.esconde_raton();
                  dibuja();
                  borra();
                  mimi.inicia_raton();
            }
            mimi.posicion (vertx3,verty3);
            setcolor(color);
            mimi.esconde_raton();
            dibuja();
            mimi.inicia_raton();
      }
      triangulo::triangulo(double x1, double y1, double x2, double y2,double  x3,
double y3)
      {
            vertx1=x1;
            vertx2=x2;
            vertx3=x3;
            verty1=y1;
            verty2=y2;
            verty3=y3;
      }

      double triangulo::area(void) const
      {
            double x21,y21,x31,y31;
            x21= vertx2 - vertx1;
            y21= verty2 - verty1;
            x31= vertx3 - vertx1;
            y31= verty3 - verty1;
            return( fabs(y21*x31-x21*y31) /2.0 );
      }
```

```
void triangulo::datos(void) const
{
      cout << endl << "Triangulo:";
      cout << endl << "\tCoordenadas:";
      cout << endl << "\t\tVertice 1: ("<< vertx1<<","<<verty1<<")";
      cout << endl << "\t\tVertice 2: ("<< vertx2<<","<<verty2<<")";
      cout << endl << "\t\tVertice 3: ("<< vertx3<<","<<verty2<<")";
}

void triangulo::dibuja(void) const
{
      int coord[8]=
      {
            verty1,vertx1,
            verty2,vertx2,
            verty3,vertx3,
            verty1,vertx1
      };

      if (graficos_ok())
            /* dibuja un tringulo */
            drawpoly(4,coord);
}
void triangulo::borra(void) const
{
      int coord[8]=
      {
            verty1,vertx1,
            verty2,vertx2,
            verty3,vertx3,
            verty1,vertx1
      };

      int color;

      color = getcolor();
      setcolor(getbkcolor());
      if (graficos_ok())
            drawpoly(4,coord);
      setcolor(color);

}

//Metodos de la clase linea

linea::linea()
{
      int color;
      // Mientras no se pulse el boton derecho
      while (mimi.boton() != 1);
      mimi.posicion (supizqx,supizqy);
      color = getcolor();
      setcolor(PREVIO);
```

```
        while (mimi.boton() == 1)
        {
              mimi.posicion (infderx,infdery);
              mimi.esconde_raton();
              dibuja();
              borra();
              mimi.inicia_raton();
        }
        mimi.posicion (infderx,infdery);
        setcolor(color);
        mimi.esconde_raton();
        dibuja();
        mimi.inicia_raton();
};
linea::linea(double x1, double y1 ,double x2 ,double y2)
{
        supizqx=x1;
        supizqy=y1;
        infderx=x2;
        infdery=y2;
};
double linea::area(void) const
{
        return (0);
};
void linea::dibuja(void) const
{

        if (graficos_ok())
              /* dibuja una linea */
              line(supizqy,supizqx,infdery,infderx);
};
void linea::datos(void)  const
{
        cout << endl << "Linea:";
        cout << endl << "\tCoordenadas:";
        cout << endl << "\t\tVertice 1: ("<< supizqx<<","<<supizqy<<")";
        cout << endl << "\t\tVertice 2: ("<< infderx<<","<<infdery<<")";

};

void linea::borra(void)  const
{
        int color;
        color = getcolor();
        setcolor(getbkcolor());

        if (graficos_ok())
              /* dibuja una linea */
              line(supizqy,supizqx,infdery,infderx);
        setcolor(color);
}
// void linea::mover(int,int)  const;
```

```
//Metodos de la clase punto

punto::punto()
{
      // Mientras no se pulse el boton derecho
      if (mimi.boton() == 1)
      {
            mimi.posicion (x,y);
      }
      else
      {
            while (mimi.boton() != 1);
            mimi.posicion (x,y);
      }
      mimi.esconde_raton();
      dibuja();
      mimi.inicia_raton();
};
punto::punto(double x1, double y1 )
{
      x=x1;
      y=y1;
};
void punto::coloca(double x1, double y1 )
{
      x=x1;
      y=y1;
};
double punto::area(void) const
{
      return (0);
};
void punto::dibuja(void) const
{
      int color;
      color=getcolor();

      if (graficos_ok())
            /* dibuja un punto */
            putpixel(y,x,color);
};
void punto::datos(void)  const
{
      cout << endl << "Punto:";
      cout << endl << "\tCoordenada: ("<< x<<","<<y<<")";

};

void punto::borra(void)  const
{
      int color;
      color = getbkcolor();
```

```
            if (graficos_ok())
                  /* dibuja una linea */
                  putpixel(y,x,color);
      }
      // void linea::mover(int,int)  const;

#endif
```

Observaciones:

El constructor del círculo obtiene las coordenadas del centro del círculo cuando el usuario oprime el botón derecho del dispositivo.

```
// FORMASLI.H
// Declaración de las clases que manejan la lista ligada de formas
// Programo: Ernesto Peñaloza Romero

//Cada elemento de la lista de formas ligadas
// contiene un puntero a la siguiente forma
#ifndef FORMASLI_H
      #define FORMASLI_H 1

      // estas dos clases son colaboradoras
      // la clase elemento_lista contiene información de un nodo
      class listaformas;
      class elemento_lista
      {
            public:
                  friend listaformas;
                  elemento_lista(forma *fig, elemento_lista *prox)
                  {
                        figura = fig;
                        siguiente = prox;
                  }
            private:
                  forma *figura;                // figura del nodo actual
                  elemento_lista *siguiente;  // puntero al siguiente nodo
      };

      // Lista de formas.
      // cada nodo es un ELEMENTO_LISTA
      class listaformas
      {
            public:
                  listaformas()
                  {
                        cabeza= new elemento_lista( (forma *)0, (elemento_lista
*)0);
                        actual = cabeza;
                  }
```

```
                // Inserta un nuevo nodo
                void inserta(forma *fig)
                {
                      elemento_lista *nodo;
                      nodo = new elemento_lista(fig,cabeza);
                      if (nodo !=(elemento_lista *)0)
                            cabeza = nodo;
                      else
                      {
                            printf("\a- Memoria agotada !");
                            getch();
                      }
                      actual=cabeza;
                }
                // Coloca el puntero en la cabeza de lista
                void inicio()
                {
                      actual = cabeza;
                }
                // Obtiene el puntero a la figura actual
                forma *activo()
                {
                      forma *nodo;
                      nodo = actual->figura;
                      return (nodo);
                }

                // Obtiene un puntero a la siguiente figura
                forma *siguiente();
                void borra(forma *fig); // Borra la figura especificada

          private:
                elemento_lista *cabeza, *actual;

    };

#endif


// FORMASLI.H
// Declaración de las clases que manejan la lista ligada de formas
// Programo: Ernestoi Penaloza Romero

//Cada elemento de la lista de formas ligadas
// contiene un puntero a la siguiente forma
#ifndef FORMASLI_H
      #define FORMASLI_H 1

      // estas dos clases son colaboradoras
      // la clase elemento_lista contiene información de un nodo
      class listaformas;
```

```
class elemento_lista
{
      public:
            friend listaformas;
            elemento_lista(forma *fig, elemento_lista *prox)
            {
                  figura = fig;
                  siguiente = prox;
            }
      private:
            forma *figura;                // figura del nodo actual
            elemento_lista *siguiente;  // puntero al siguiente nodo
};

// Lista de formas.
// cada nodo es un ELEMENTO_LISTA
class listaformas
{
      public:
            listaformas()
            {
                  cabeza= new elemento_lista( (forma *)0, (elemento_lista
*)0);
                  actual = cabeza;
            }
            // Inserta un nuevo nodo
            void inserta(forma *fig)
            {
                  elemento_lista *nodo;
                  nodo = new elemento_lista(fig,cabeza);
                  if (nodo !=(elemento_lista *)0)
                        cabeza = nodo;
                  else
                  {
                        printf("\a- Memoria agotada !");
                        getch();
                  }
                  actual=cabeza;
            }
            // Coloca el puntero en la cabeza de lista
            void inicio()
            {
                  actual = cabeza;
            }
            // Obtiene el puntero a la figura actual
            forma *activo()
            {
                  forma *nodo;
                  nodo = actual->figura;
                  return (nodo);
            }
```

```
                // Obtiene un puntero a la siguiente figura
                forma *siguiente();
                void borra(forma *fig); // Borra la figura especificada

        private:
                elemento_lista *cabeza, *actual;

};

#endif


// FORMASLI.H
// Declaraci¢n de las clases que manejan la lista ligada de formas
// Programo: Ernesto Penaloza Romero

//Cada elemento de la lista de formas ligadas
// contiene un puntero a la siguiente forma
#ifndef FORMASLI_H
      #define FORMASLI_H 1

      // estas dos clases son colaboradoras
      // la clase elemento_lista contiene informaci¢n de un nodo
      class listaformas;
      class elemento_lista
      {
            public:
                  friend listaformas;
                  elemento_lista(forma *fig, elemento_lista *prox)
                  {
                        figura = fig;
                        siguiente = prox;
                  }
            private:
                  forma *figura;              // figura del nodo actual
                  elemento_lista *siguiente;  // puntero al siguiente nodo
      };

      // Lista de formas.
      // cada nodo es un ELEMENTO_LISTA
      class listaformas
      {
            public:
                  listaformas()
                  {
                        cabeza= new elemento_lista( (forma *)0, (elemento_lista
*)0);
                        actual = cabeza;
                  }
```

```
                // Inserta un nuevo nodo
                void inserta(forma *fig)
                {
                      elemento_lista *nodo;
                      nodo = new elemento_lista(fig,cabeza);
                      if (nodo !=(elemento_lista *)0)
                            cabeza = nodo;
                      else
                      {
                            printf("\a- Memoria agotada !");
                            getch();
                      }
                      actual=cabeza;
                }
                // Coloca el puntero en la cabeza de lista
                void inicio()
                {
                      actual = cabeza;
                }
                // Obtiene el puntero a la figura actual
                forma *activo()
                {
                      forma *nodo;
                      nodo = actual->figura;
                      return (nodo);
                }

                // Obtiene un puntero a la siguiente figura
                forma *siguiente();
                void borra(forma *fig); // Borra la figura especificada

          private:
                elemento_lista *cabeza, *actual;

    };

#endif


// BOTON.CPP
// Rutinas para desplegar un bot$n.
// Programo: Ernesto Penaloza Romero.
// Ultima actualizaci$n 3 de junio de 1993

#ifndef BOTON_H
      #define BOTON_H

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

```
// Pinta un bot$n en la pantalla
void boton(int ren_1,int col_1,int ren_2,int col_2)
{
      int MaxX,MaxY;
      int cursor,color_cara;
      int color;
      int numcolores = getmaxcolor();
      // Esta estructura se encuentra definida en graphics.h
      struct fillsettingstype infor_lleno;

      MaxX = col_2;
      MaxY = ren_2;

      // Define color del boton dependiendo del numero de
      // colores disponibles
      color=getcolor();
      if (numcolores > 2)
            color_cara=CYAN;
      else
            color_cara=WHITE;

      // obtiene caracteristicas de llenado de figuras
      getfillsettings(&infor_lleno);
      setfillstyle(SOLID_FILL,color_cara);

      // coloca la cara del boton
      bar(col_1,ren_1,col_2,ren_2);

      // Numero de lineas que conforman la sombra del boton
      for (cursor=0;cursor<2;cursor++)
      {
            // lado superior brillante del boton
            if (numcolores > 2)
                  setcolor(LIGHTCYAN);
            line(col_1+cursor,ren_1+cursor,MaxX-cursor,ren_1+cursor);
            line(col_1+cursor,ren_1+cursor,col_1+cursor,MaxY-cursor);

            // lado obscuro del boton
            if (numcolores > 2)
                  setcolor(DARKGRAY);
            else
                  setcolor(BLACK);
            line(col_1+cursor,MaxY-cursor,MaxX-cursor,MaxY-cursor);
            line(MaxX-cursor,ren_1+cursor,MaxX-cursor,MaxY-cursor);
      }
      // restaura condiciones iniciales
      setfillstyle(infor_lleno.pattern,infor_lleno.color);
      setcolor(color);

}
```

```
// Cuando se oprime un boton del ratón, se borran las lineas de sombra
void click(int ren_1,int col_1,int ren_2,int col_2)
{
      int MaxX,MaxY;
      int cursor,color_cara;
      int color;
      struct fillsettingstype infor_lleno;

      MaxX = col_2;
      MaxY = ren_2;

      // color del boton
      color=getcolor();
      if (getmaxcolor() > 2)
            color_cara=CYAN;
      else
            color_cara=WHITE;

      // opciones previas
      getfillsettings(&infor_lleno);
      setfillstyle(SOLID_FILL,color_cara);


      setcolor(color_cara);
      for (cursor=0;cursor<2;cursor++)
      {
            // lado superior brillante del boton
            line(col_1+cursor,ren_1+cursor,MaxX-cursor,ren_1+cursor);
            line(col_1+cursor,ren_1+cursor,col_1+cursor,MaxY-cursor);

            // lado obscuro del boton
            line(col_1+cursor,MaxY-cursor,MaxX-cursor,MaxY-cursor);
            line(MaxX-cursor,ren_1+cursor,MaxX-cursor,MaxY-cursor);
      }
      setfillstyle(infor_lleno.pattern,infor_lleno.color);
      setcolor(color);
}

// Restaura las sombras del boton
void noclick(int ren_1,int col_1,int ren_2,int col_2)
{
      int MaxX,MaxY;
      int cursor,color_cara;
      int color;
      struct fillsettingstype infor_lleno;

      MaxX = col_2;
      MaxY = ren_2;

      color=getcolor();
      if (getmaxcolor() > 2)
            color_cara=CYAN;
      else
            color_cara=WHITE;
```

```
            // color del boton
            getfillsettings(&infor_lleno);
            setfillstyle(SOLID_FILL,color_cara);

            for (cursor=0;cursor<2;cursor++)
            {
                  // lado superior brillante del boton
                  setcolor(LIGHTCYAN);
                  line(col_1+cursor,ren_1+cursor,MaxX-cursor,ren_1+cursor);
                  line(col_1+cursor,ren_1+cursor,col_1+cursor,MaxY-cursor);

                  // lado obscuro del boton
                  setcolor(DARKGRAY);
                  line(col_1+cursor,MaxY-cursor,MaxX-cursor,MaxY-cursor);
                  line(MaxX-cursor,ren_1+cursor,MaxX-cursor,MaxY-cursor);
            }
            setfillstyle(infor_lleno.pattern,infor_lleno.color);
            setcolor(color);
      }

#endif

// MENU.CPP :Esta subrutina pinta los botones que constituyen
// el menu de figuras que GARABUJOS puede dibujar
// Programo: Ernesto Penaloza Romero
// Ultima actualizaci¢n: 3 de junio de 1993

#define num_fig 5   // numero de figuras que GARABUJOS puede hacer

int menu(void)
{

      char buff[5];
      int opcion,ancho,contador,diff;
      int x,y,temp;
      int color,color_fondo;

      // Establece un raton
      raton unico;

      // obtiene el ancho del boton
      ancho = getmaxy()/num_fig;
      if ( ancho > 40)
            ancho = 40;
      // proporci¢n del ¡cono en el boton
      diff = ancho / 5;

      //Almacena colores actuales
      color=getcolor();
      color_fondo = getbkcolor();
      setcolor(color_fondo);
```

```
// pinta botones
setfillstyle(SOLID_FILL, color_fondo);
for( contador =0; contador < num_fig; contador++)
{
      boton(ancho*contador,0, (1+contador)*ancho-1,ancho);
}
ancho = ancho/2;

// coloca los iconos en cada boton
/* circulo*/
circle(ancho,ancho,ancho-diff);
/* rectangulo */
ancho = ancho*2;
rectangle(diff,ancho+diff,ancho-diff,(ancho*2)-diff);
/* triangulo */
line (diff,ancho*2+diff,diff,ancho*3-diff);
line (diff,ancho*2+diff,ancho-diff,(ancho*3)-diff);
line (diff,ancho*3-diff,ancho-diff,(ancho*3)-diff);
/* linea */
line (diff,ancho*3+diff,ancho-diff,(ancho*4)-diff);
/* linea a mano libre */
temp=0;
for (contador=0; contador<ancho-diff*2; contador++)
{
      temp=random(2)+temp;
      putpixel(diff*2+temp,ancho*4+diff+contador,color_fondo);
}

// Espera a que se pulse el boton derecho
while  (unico.boton() !=BO_DERECHO);
opcion=-1;

// Espera a que se pulse una opci¢n dentro del menu :
// Mientras( opcion no valida y boton derecho oprimido)
while ((opcion < 1 || opcion >num_fig)|| unico.boton() ==BO_DERECHO)
{
      unico.posicion(x,y);
      // Si se oprime el boton derecho y se encuentra en el
      // area de los botones
      if (y <= ancho && unico.boton()==BO_DERECHO)
      {
            // Obtiene cual boton de menu se est  oprimiendo
            temp = x/ancho+1;

            // Si el boton de menu seleccionado ha cambiado
            if (temp !=opcion)
            {
                  unico.esconde_raton();
                  // deprime el boton de menu seleccionado con anterioridad
                  if ((opcion >= 1 && opcion <= num_fig))
                        noclick(ancho*(opcion-1),0,opcion*ancho-1,ancho);
```

```
                        // oprime nuevo boton de menu
                        opcion=temp;
                        click(ancho*(opcion-1),0, opcion*ancho-1,ancho);
                        unico.inicia_raton();
                    }
                }
        }

        // Borra botones de menu
        unico.esconde_raton();
        setfillstyle(SOLID_FILL, color_fondo);
        for( contador =0; contador < num_fig; contador++)
        {
                bar(0,ancho*contador,ancho,(1+contador)*ancho-1);
        }

        //Reestablece los colores y atributos previos
        setfillstyle(SOLID_FILL, color);
        setcolor(color);
        unico.posicion(x,y);
        return (opcion);
}

/*
        GARABUJO.CPP

        Universidad Nacional Autonoma de Mexico
        Escuela Nacional de Estudios Profesionales
        Unidad Academica Aragon

        GARABUJOS. Ver 1.0

                Programo: Ernesto Penaloza Romero

                Fecha de creacion: 5 de abril de 1993
                Fecha de liberaci¢n: 28 de mayo de 1993
                Fecha de ultima actualizaci¢n: 2 de junio de 1993

*/
#include "stdio.h"
#include "iostream.h"
#include "raton.cpp"
#include "graflib.cpp"
#include "formas.h"
#include "formas.cpp"
#include "formasli.h"
#include "formasli.cpp"
#include "boton.cpp"
#include "entrada.cpp"
#include "menu.cpp"

#define ESC 27          // Tecla ESCAPE en ASCII
#define ENTER 13    // Tecla ENTER  en ASCII
```

```
void main(void)
{
      // Instancias de clase
      pantalla grafica;          // Crea un pantalla grafica
      listaformas dibujo;        // Lista ligada de formas
      raton micky;               // Instancia del raton
      int i,x,y,button,opcion;
      int tecla;
      int color;

      color= WHITE; //Color inicial de las figuras
      setcolor(color);
      if (getmaxcolor()>2)
            setbkcolor(BLUE);
      micky.inicia_raton();
//    entrada();
      micky.coloca_raton(getmaxy()/2,getmaxx()/2);

      while (button != BO_AMBOS)
      {

            gotoxy(1,25);
            printf("Opci¢n: ");

            // ¨Qu, boton ha sido oprimido ?
            button = micky.boton();
            switch(button)
            {
                  case BO_NINGUNO:
                        printf("Ninguna        ");
                        break;

                  // Inserta figura
                  case BO_DERECHO:
                        printf("Inserta ");
                        opcion= menu();
                        switch(opcion)
                        {
                              // Inserta circulo
                              case 1:
                                    printf("circulo   ");
                                    dibujo.inserta(new circulo);
                                    break;

                              // Inserta rectangulo
                              case 2:
                                    printf("rectangulo");
                                    dibujo.inserta(new rectangulo);
                                    break;
```

```
                // Inserta triangulo
                case 3:
                      printf("triangulo");
                      dibujo.inserta(new triangulo);
                      break;

                //Inserta una recta
                case 4:
                      printf("recta    ");
                      dibujo.inserta(new linea);
                      break;

                // Trazos a mano alzada
                case 5:
                      printf("trazos   ");
                      {
                            // Variables v lidas solo en
                            // en este bloque
                            int x=0,y=0;
                            int xtemp,ytemp;

                            // Aparece raton y espera click en
                            // el boton derecho
                            micky.inicia_raton();
                            while (micky.boton() != BO_DERECHO);
                            while (micky.boton() == BO_DERECHO)
                            {
                                  micky.posicion(xtemp,ytemp);

                                  // Si la posici¢n del cursor ha
                                  //cambiado inserta el nuevo punto
                                  if (x!=xtemp || y!=ytemp)
                                  {
                                        x=xtemp;
                                        y=ytemp;
                                        dibujo.inserta(new punto);
                                  }
                            }
}

                      break;

                defualt:
                      break;
}

// limpia pantalla e iinicia lista
//para dibujar la figura completa
cleardevice();
dibujo.inicio();
```

```
            // mientras existan figuras en la lista
            while(dibujo.activo() != (forma *)0)
            {
                  dibujo.activo()->dibuja();
                  dibujo.siguiente();
            }

            break;

      // Se desea borrar
      case BO_IZQUIERDO:
      printf("Borrando");
            gotoxy(1,1);
            printf("TAB  : Escoge\nESC  : Sale\nENTER: Borra");

            // Inicia la lista para poder elegir cada figura
            dibujo.inicio();
            tecla='\t';
            while(dibujo.activo() != (forma *)0 && tecla != ESC)
            {
                  // Sobresalta la figura activa
                  setcolor(PREVIO);
                  dibujo.activo()->dibuja();
                  //Obten comando
                  tecla=getch();
                  // Si la figura es aceptada, borrala de la lista
                  if (tecla == ENTER)
                  {
                        dibujo.activo()->borra();
                        dibujo.borra(dibujo.activo());
                  }
                  else
                  {
                        // sino, apunta a la siguiente figura
                        setcolor(color);
                        dibujo.activo()->dibuja();
                        dibujo.siguiente();
                  }
            }

            // Dibujamos todo de nuevo
            setcolor(color);
            cleardevice();
            dibujo.inicio();
            while(dibujo.activo() != (forma *)0)
            {
                  dibujo.activo()->dibuja();
                  dibujo.siguiente();
            }
            break;
```

```
            // Opcion de salida
            case BO_AMBOS:
                        printf("SALIDA     ");
                        break;
            }
  }
  micky.apaga_raton();
}
```

II.6 Constructores y destructores

Cuando un programador declara una variable en la forma tradicional, el compilador crea el espacio necesario para alojar al dato en cuestión. Cuando se crea un objeto, se realiza una acción similar.

En general un constructor se define con el mismo nombre de la clase. Un constructor no devuelve ningún tipo. El compilador no aceptará la inclusión de la palabra **void**.

En cambio un constructor puede ser definido para distintos tipos de argumentos. El argumento correcto será establecido en base al tipo de argumentos con los cuales sea llamado. Pueden definirse tantos constructores como sea necesario. La única condición es que todas las listas de argumentos sean efectivamente distintas.

En ocasiones esta diferencia puede no existir si no se tiene cuidado. Por ejemplo, en los siguientes constructores no existe diferencia alguna y el compilador marcará un error al no poder distinguir entre ambos.

```
ejemplo::ejemplo();
ejemplo::ejemplo(int i=0, int j=0);
```

La confusión por parte del compilador proviene del hecho de que el segundo constructor tiene argumentos por omisión. En el momento en que se defina una instancia de la clase ejemplo sin argumentos el compilador puede hacer dos cosas: En la primera deberá llamar al constructor vacio y ejecutar su método; la segunda opción deberá llamar al constructor con argumentos por omisión, asignar los valores que se especifican en la lista y continuar con la ejecución de este método.

La construcción de un objeto con diversos argumentos resulta muy beneficioso para el usuario de la clase ya que le permite una gran flexibilidad en la declaración de una nueva instancia de clase. Considerense las distintas formas en que puede declarase una cadena de caractéres.

```
cadena variable= "Esta es una cadena";
cadena var_2=variable;
cadena *linea= new cadena[80];
cadena  nombre
```

Los constructores de clase debrán ser capaces de aceptar estas distintas formas de declarar una cadena y crear una nueva instancia:

```
cadena();
cadena(size_t longitud);
cadena(const char *puntero_cadena);
cadena(const cadena& referencia);
```

Un destructor también recibe el nombre de la clase a la cual pertenece pero le antecede el simbolo ~. Los destructores son especialmente importantes cuando se crean objetos con new ya que con esto de ha realizado una petición de memoria. Dicha memoria debe ser devuelta al sistema antes de que el programa termine o cuando se determine que la vida del objeto ha concluido.

Cuando una clase hereda de otra el orden de llamada de los constructores y destructores es importante. Siempre se llama primero al constructor de la clase base. Observese el listado III.6.1 que muestra una serie de clases. La clase b hereda en forma múltiple tanto de la clase base como de la clase a. La clase c hereda de b. El programa principal solo crea una instancia de c y no realiza llamadas explicitas a los destructores

```
// Listado III.6.1
#include <iostream.h>
#include <conio.h>

class base
{
public:
      base::base(int x = 0 )
      {
            cout << endl<<"Constructor base::base(" << x << ") ";
      }
      ~base()
      {
            cout <<"\n ¡ Destruyendo  base!";
      }
private:
      int x;
};

class a
{

public:
      a(int x)
      {
            cout <<endl<< "Constructor a::a (" << x<< ") " ;
      }
      ~a()
      {
            cout <<"\n ¡ Destruyendo  a !";
      }
};
class b: public base, public a
{
public:
      b(int x): base(x-2),a(x-1)
      {
            cout <<endl<< "Constructor b::b (" << x<< ") " ;
      }
```

```
      ~b()
      {
            cout <<"\n ¡ Destruyendo  b !";
      }
};

class c: public b
{
public:
      c(int x): b(x-1)
      {
            cout <<endl<< "Constructor c::c (" << x<< ") ";
      }
      ~c()
      {
            cout <<"\n ¡ Destruyendo  c!";
      }
};

main()
{
      clrscr();
      c(4);
}
```

Resultados:

```
Constructor base::base(1)
Constructor a::a (2)
Constructor b::b (3)
Constructor c::c (4)
 ¡ Destruyendo  c!
 ¡ Destruyendo  b !
 ¡ Destruyendo  a !
 ¡ Destruyendo  base!
```

En esta versión del ejemplo se observa que las llamadas a los constructores se realizan comenzando con las clases bases hasta finalizar con la clase heredera. Los destructores no son llamados en forma explicita pero aún así son llamados cuando el objeto cesa su existencia. Los destructores son llamados en el orden inverso en el cual fueron llamados los constructores.

Un segundo punto de mención se encuentra en la clase b la cual hereda en forma múltiple de a y de la clase **base,** los constructores fueron llamados en base al orden con el cual fueron definidos en la línea de definición de clase. El orden de las llamadas en la línea de definición del constructor no es trascendente.

Cuando se crea una instancia heredera también se puede realizar una llamada implícita. Presentaremos una nueva versión del mismo ejemplo. Observense las diferencias existentes en el código del constructor de la clase b y la llamada implícita en el constructor de la clase d, nueva en este ejemplo.

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

class base
{
public:
      base::base(int x = 0 )
      {
            cout << endl<<"Constructor base::base(" << x << ") ";
      }
      ~base()
      {
            cout <<"\n ¡ Destruyendo  base!";
      }
private:
      int x;
};

class a
{

public:
      a(int x)
      {
            cout <<endl<< "Constructor a::a (" << x<< ") " ;
      }
      ~a()
      {
            cout <<"\n ¡ Destruyendo  a !";
      }
};
class b: public base, public a
{
public:
      b(int x): a(x-1),base(x-2)
      {
            cout <<endl<< "Constructor b::b (" << x<< ") " ;
      }
      ~b()
      {
            cout <<"\n ¡ Destruyendo  b !";
      }
};

class c: public b
{
      public:
      c(int x): b(x-1)
      {
            cout <<endl<< "Constructor c::c (" << x<< ") ";
      }
```

```
      ~c()
      {
            cout <<"\n ¡ Destruyendo  c!";
      }
};
class d: public base
{
public:
      d()
      {
            cout <<endl << "Constructor d::d";
      }
      ~d()
      {
            cout <<"\n ¡ Destruyendo d!";
      }
};

main()
{
      clrscr();
      c c1(4);
      d d1;
}
```

Resultados:

```
Constructor base::base(1)
Constructor a::a (2)
Constructor b::b (3)
Constructor c::c (4)
Constructor base::base(0)
Constructor d::d
 ¡ Destruyendo d!
 ¡ Destruyendo  base!
 ¡ Destruyendo  c!
 ¡ Destruyendo  b !
 ¡ Destruyendo  a !
 ¡ Destruyendo  base!
```

La clase **b** se encuentra heredando de las clases **base** y **a**. Observe con atención que el orden en el cual se encuentran definidas en la definición del constructor es :

```
b(int x): a(x-1),base(x-2)
```

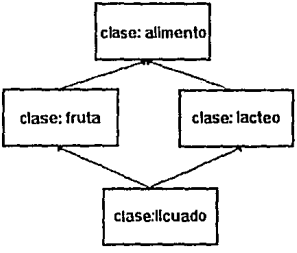
sin embargo las llamadas a los constructores fueron realizadas en base al orden en el cual se definió la herencia en de la clase:

```
class b: public base, public a.
```

Por otra parte la clase **d** hereda de la clase **base** pero no realiza una llamada explícita al contructor de dicha clase, no obstante, la llamada es ejecutada.

CONSTRUCTORES EN CLASES VIRTUALES

Una de las razones más importantes para la detección de una clase virtual es un problema que se puede presentar cuando se realiazan las llamadas a las clases bases. Consideremos el siguiente ejemplo: dentro de los alimentos encontramos a las frutas y a los lácteos. Las clases fruta y lacteos son derivadas de la clase alimento. A su vez, un licuado de fresa es una combinación de un vaso de leche (instancia de lacteo) y unas fresas (instancias de fruta).



Veamos como codificaríamos estas relaciones jerarquicas y su repercusión en los constructores:

```
// Ejemplo: Ilustra la necesidad de definir correctamente
// las clases virtuales

#include <iostream.h>


class alimento
{
      public:
            alimento()
            {
                  cout << "Es un alimento" << endl;
            }
};

class fruta: public alimento
{
      public:
            fruta()
            {
                  cout << "Es una fruta" << endl;
            }
};
```

```
class lacteo: public alimento
{
      public:
            lacteo()
            {
                  cout << "Es un lácteo" << endl;
            }
};

class licuado: public fruta, public lacteo
{
      public:
            licuado()
            {
                  cout << "Esto es un licuado" << endl;
            }
};

void main()
{
      licuado de_fresa;
}
```

Resultados:

```
Es un alimento
Es una fruta
Es un alimento
Es un lácteo
Esto es un licuado
```

Como se puede apreciar, se realiza una llamada duplicada a la clase alimento. La razón de ello se encuentra en que tanto la clase `frutas` como la clase `lacteos` heredan de la misma clase común. El resultado final de ello es que las instancias de la clase `licuado` poseerán en forma duplicada las características heredadas de la superclase `alimento`.

Para evitar ese problema, no detectado por el compilador, es necesario indicar al compilador que existen clases virtuales:

```
// Ejemplo: Ilustra la necesidad de definir correctamente
// las clases virtuales

#include <iostream.h>
```

```
class alimento
{
      public:
            alimento()
            {
                  cout << "Es un alimento" << endl;
            }
};

class fruta: public virtual alimento
{
      public:
            fruta()
            {
                  cout << "Es una fruta" << endl;
            }
};

class lacteo: public virtual alimento
{
      public:
            lacteo()
            {
                  cout << "Es un l cteo" << endl;
            }
};

class licuado: public fruta, public lacteo
{
      public:
            licuado()
            {
                  cout << "Esto es un licuado" << endl;
            }
};

void main()
{
      licuado de_fresa;
}
```

Resultado:

```
Es un alimento
Es una fruta
Es un lácteo
Esto es un licuado
```

III.7 Sobrecarga de operadores

Sobrecargar un operador significa que se le dotará de un método para que pueda trabajar con tipo de datos distintos a los que normalmente se suele manejar con ellos. Se puede sobrecargar un mismo operador con múltiples funciones.

FUNCIONES friend

Una función puede ser amiga (friend) de una clase pero no necesariamente tiene que ser una función miembro. Una función friend tiene acceso a todos los datos privados de una clase a la cual se le declara como amiga.

Para declarar que una función es amiga de otra se le antepone la palabra reservada **friend**.

OPERADOR this

Hasta este momento hemos utilizado los miembros de una clase colocando el nombre del miembro sin hacer ninguna referencia a la clase. En C++ se realiza una definición automática de dicha referencia por medio del operador `this`. Este operador es exclusivo de C++.

El operador `this` es pasado como un argumento adicional y es un puntero al objeto para el cual la función haya sido invocada. Esto significa que el puntero this contiene la dirección del primer elemento del objeto. Esta dirección será necesariamente la del primer dato contenido en la declaración de clase. Cada compilador tiene la libertad de decidir cuales variables deben ser localizadas primero, sin tomar en cuenta el orden de aparición en la declaración de clase. La palabra `this` puede ser leída como *la dirección inicial del objeto para la que fue llamada la función*

Cada miembro de una clase tiene en forma implícita la forma `this -> miembro`. Tal y como ya se ha mencionado, no es necesario codificar en forma explícita está referencia. El compilador la realiza por nosotros.

Podemos retomar a una clase cliente un objeto al escribir una sentencia como la siguiente:

```
return *this
```

SINTAXIS DE LA SOBRECARGA

La sintaxis de la sobrecarga es muy sencilla.
Para una función friend:

tipo operator @ (lista de argumentos) {cuerpo de la función}

Para una función miembro:

tipo nombre_clase :: operator @ (argumento) {cuerpo de la funcion}

Cuando se define una sobrecarga a través de una función miembro, se asume que se trata de la llamada de un operador del tipo de la clase que realiza la sobrecarga con un operador de un tipo definido por el programador. Esto significa que una sobrecarga por medio de una clase miembro solo requiere de un argumento.

Cuando se realiza una sobrecarga por medio de una función friend es necesario utilizar dos argumentos. La función friend no forma parte de la clase pero puede accesarla en forma total.

Existe una buena razón para realizar una sobrecarga con una función friend en lugar de utilizar una función miembro. Cuando se realiza la llamada al método de sobrecarga con una función miembro, esta se ejecuta tomando el tipo de dato del operando izquierdo.

Programa:

Un número complejo consta de una parte real y una parte imaginaria. A un número complejo se le puede representar en un plano cartesiano con un eje real y un eje imaginario. La notación que se basa en dicho plano recibe el nombre de notación en *forma cartesiana*. Su notación es:

numero = real + imag *j*

Donde:

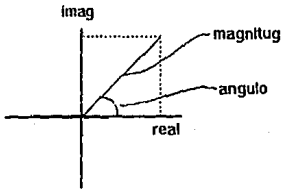
real	Representa la parte real de un número complejo
imag	Representa la parte imaginaria del número
j	Es el símbolo para identificar a la parte imaginaria

La *forma polar* representa a un numero complejo como un vector en el mismo plano cartesiano. El vector es denotado por su magnitud y el ángulo que forma con el eje real. Su notación es:

numero = magnitud ∠ angulo

Donde:

magnitud	Representa la magnitud del vector
∠	Es el símbolo que denota el ángulo
angulo	Es el valor del ángulo del vector



Escribir un programa que sobrecarge los operadores básicos para números complejos en su forma polar.

Solución:

```
// compejo.h
// Archivo de encabezado para la sobrecarga de operadores
// para numeros complejos en forma polar
// 6/abr/1993
// Ernesto Peñaloza Romero
```

```
#if !defined _COMPLEJO_H
      #define _COMPLEJO_H     1

      #include <math.h>
      #include <iostream.h>

      #define mod(x,y) sqrt(x*x+y*y)

      class compl_polar
      {


      public:
            // constructores
            compl_polar(double& norma, double& ang);
            compl_polar();

            // Manipuladores, obtienen informacion privada
             friend double real(compl_polar&);    // Parte real
             friend double imag(compl_polar&);    // Parte imaginaria
             friend double norma(compl_polar&);   // Modulo
             friend double ang(compl_polar&);     // Angulo
             friend compl_polar conj(compl_polar&);  // Complejo conjugado

            // Operadores y Funciones Binarias
             friend compl_polar operator+(compl_polar&, compl_polar&);
             friend compl_polar operator+(double, compl_polar&);
             friend compl_polar operator+(compl_polar&, double);
             friend compl_polar operator-(compl_polar&, compl_polar&);
             friend compl_polar operator-(double, compl_polar&);
             friend compl_polar operator-(compl_polar&, double);
             friend compl_polar operator*(compl_polar&, compl_polar&);
             friend compl_polar operator*(compl_polar&, double);
             friend compl_polar operator*(double, compl_polar&);
             friend compl_polar operator/(compl_polar&, compl_polar&);
             friend compl_polar operator/(compl_polar&, double);
             friend compl_polar operator/(double, compl_polar&);
             compl_polar operator=(compl_polar);
             friend int operator==(compl_polar&, compl_polar&);

             //Sobrecarga de la funcion de salida
             void print(ostream& os)const;

      // Implantacion
      private:
            double modulo;
            double angulo;
      };
#endif
```

```
// Implantacion de los metodos de la clase complejos
// 6/abr/1993
// Ernesto Peñaloza Romero

#include <iostream.h>

// Sobrecarga de las funciones de entrada y salida
ostream& operator<<(ostream& os, compl_polar& z);
istream& operator>>(istream& is, compl_polar& z);

// Funciones en linea compl_polar

inline compl_polar::compl_polar(double& norma, double& ang)
{
            modulo = norma;
            angulo = ang;
}
inline compl_polar::compl_polar()
{
      // Creacion sin argumentos
      modulo = angulo = 0;
}

// Parte real
inline  double real(compl_polar& z)
{
      return (z.modulo* cos(z.angulo));
}

// Parte imaginaria
double imag(compl_polar& z)
{
      return(z.modulo*sin(z.angulo));
}

// Modulo
double norma(compl_polar& z)
{
      return (z.modulo);
}
// Angulo
double ang(compl_polar& z)
{
      return (z.angulo);
}

// Complejo conjugado
compl_polar conj(compl_polar& z)
{
      return compl_polar(z.modulo,-z.angulo);
}
```

```
// Operadores
inline  compl_polar operator+(compl_polar& z1, compl_polar& z2)
{
      double res_real;
      double res_imag;

      res_real=real(z1)+real(z2);
      res_imag=imag(z2)+imag(z2);
return compl_polar( mod(res_real,res_imag),atan(res_imag/res_real));
}

inline  compl_polar operator+(double real, compl_polar& z)
{
      return compl_polar (real+z.modulo,z.angulo);
}

inline  compl_polar operator+(compl_polar& z, double real)
{
      return compl_polar (real+z.modulo,z.angulo);
}

inline  compl_polar operator-(compl_polar& z1, compl_polar& z2)
{
      double res_real;
      double res_imag;

      res_real=real(z1)-real(z2);
      res_imag=imag(z1)-imag(z2);
return compl_polar( mod(res_real,res_imag),atan(res_imag/res_real));
}

inline  compl_polar operator-(double real , compl_polar& z)
{
      return compl_polar (real-z.modulo,z.angulo);
}

inline  compl_polar operator-(compl_polar& z, double real)
{
      return compl_polar (z.modulo-real,z.angulo);
}

inline  compl_polar operator*(compl_polar& z1, compl_polar& z2)
{
      return compl_polar (z1.modulo*z2.modulo,z1.angulo+z2.angulo);
}

inline  compl_polar operator*(compl_polar& z, double real)
{
      return compl_polar (real*z.modulo,z.angulo);
}
```

```
inline  compl_polar operator*(double real , compl_polar& z)
{
      return compl_polar (real*z.modulo,z.angulo);
}

inline  compl_polar operator/(compl_polar& z1 , compl_polar& z2)
{
      return compl_polar (z1.modulo/z2.modulo,z1.angulo-z2.angulo);
}

inline  compl_polar operator/(compl_polar& z, double real)
{
      return compl_polar (z.modulo/real,z.angulo);
}

inline  compl_polar operator/(double real , compl_polar& z)
{
      return compl_polar (real/z.modulo,-z.angulo);
}

inline compl_polar compl_polar::operator=(compl_polar z)
{
      modulo=z.modulo;
      angulo=z.angulo;
      return *this;
}

inline  int operator==(compl_polar& z1 , compl_polar& z2)

{
      return (z1.modulo == z2.modulo && z1.angulo == z2.angulo);
}

// Sobrecarga del operador de salida
void compl_polar::print(ostream& os) const
{
   os << modulo << "∠ " << angulo;
}

ostream& operator<<(ostream& os, compl_polar& z)
{
      z.print(os);
      return os;
}



// Uso de la clase de complejos polares
// Observese que la muestra no es exahustiva
// 6/abr/1993
// Ernesto Peñaloza Romero
```

```
#include <conio.h>
#include "complejo.h"
#include "complejo.cpp"

void main(void)
{
      compl_polar a(10,5), b(5,5);
      compl_polar c;

      clrscr();

      // Uso del operador de salida
      cout << endl << "A= "<< a;
      cout << endl << "B= "<< b;

      // Operaciones aritmeticas sobrecargadas
      cout << endl << "C=A*B = "<< a*b;
      cout << endl << "C=A/B = "<< a/b;
      cout << endl << "C=5*A = "<< 5.0*a;
      cout << endl << "C=a-b = "<< a-b;

      // Asignacion de objetos
      c=a+b;
      cout << endl << "C=A+B = "<< c;
}
```

Resultados:

```
A= 10∠ 5
B= 5∠ 5
C=A*B = 50∠ 10
C=A/B = 2∠ 0
C=5*A = 50∠ 5
C=a-b = 5∠ -1.283185
C=A+B = 10.490855∠ -1.153178
```

OPERADOR =()

Una fuente de errores difíciles de rastrear parte de la forma como es tratada la herencia de la sobrecarga de operadores. En general, la sobrecarga de todos los operadores es heredable de una clase base hacia sus clases derivadas, con la única excepción del operador de asignación =(). Esto es debido a que las clases herederas suelen definir nuevos tipos de datos y las asignaciones no se pueden realizar. Cada clase heredera debe definir sus propios constructores y destructores para el operador de asignación ya que estos no son heredables.

Una sobrecarga en un operador de asignación exige que se retorne una referencia. Este es el caso particular por el cual se implementó el paso de parámetros por referencia. Veamos el siguiente ejemplo:

```
#include <iostream.h>

clase suma
{
      int numero;
      public:
            suma ( int valor=0 )
            {
                  numero = valor;
            }
            suma operator +(suma &operando)
            {
                  return suma(numero + operando.numero);
            }
            valor()
            {
                  cout << numero);
            }
}
void main(void)
{
      suma A(5), B(6);
      suma C = A + B;

      cout << "C = A + B = " << C.valor();
}
```

En este programa el operador sobrecargado (+) es invocado por el objeto de la izquierda y toma copmo argumento al objeto de la derecha. Dicho objeto es tomado por referencia. Recuerde que en el paso por referencia solo es necesario realizar la declaración en la linea de parámetros. La sintaxis es bastante clara en el programa principal y se comprende perfectamente que un objeto suma su valor a otro.

Este mismo programa puede ser implantado por punteros. La nueva versión es mostrada a continuación.

```
#include <iostream.h>

clase suma
{
      int numero;
      public:
            suma ( int valor=0 )
            {
                  numero = valor;
            }
            suma operator +(suma *operando)
            {
                  return suma(numero + operando->numero);
            }
```

```
            valor()
            {
                  cout << numero);
            }
}
void main(void)
{
      suma A(5), B(6);
      suma C = A + &B;

      cout << "C = A + B = " << C.valor();
}
```

La función miembro de sobrecarga del operador espera que se le envíe el puntero que contiene la dirección del objeto de la derecha, lo cual nos lleva a escribir la suma con una forma bastante extraña:

```
A + &B.
```

La claridad de la notación se ha perdido. ¿Cómo puede saberse sin revisar la implantación de la clase si se está sumando el valor del objeto A y el valor del objeto B, o si se está agregando al objeto A la dirección del objeto B? En expresiones más complejas, la notación puede perderse en caminos incomprensibles para el programador.

Aunque en el caso de los operadores, podrían enviarse parámetros por valor, el operador de asignación necesita disponer de la dirección del objeto temporal que contiene el resultado de la operación para asignarlo al objeto que contendrá el resultado. Si esa dirección se obtiene por punteros, se cae en la perdida de claridad expuesta anteriormente, en cambio, al obtener la dirección por referencia, la legibilidad persiste todo el tiempo.

CAPITULO

IV

JUSTIFICACION Y ALCANCE

IV.1 El nuevo plan de estudios de la carrera de Ingeniería en computación.

Una de las frases que se han convertido en verdaderas tautologías dentro de la sociedad moderna es: *La tecnología avanza todos los días*. La computación no se excluye de este marco de continuo avance dentro de nuestro mundo en constante cambio. Sin temor a parecer exagerados es posible afirmar que en materia de computación las cosas quedan obsoletas en cuestión de meses.

Es dentro de este marco tan dinámico que el plan de estudios de la carrera de Ingeniero en computación de la Escuela Nacional de Estudios Profesionales ha sido renovado en recientes fechas. Las imperiosas necesidades que la sociedad mexicana requiere y la labor eminentemente formativa que tiene la Universidad Nacional requieren de profesionales más capacitados para enfrentar los tiempos de crecimiento y madurez que vive la nación.

Estos apuntes han surgido como una necesidad de disponer de material que de otra manera sería inaccesible a los estudiantes de la carrera dentro de nuestra Universidad, que por su caracter de pública, alberga a estudiantes de los más diversos niveles económicos. Estos apuntes han sido escritos teniendo en mente que, a través de su publicación por parte de la ENEP, sean de utilidad como auxiliares didácticos para el profesor de asignatura.

Debido a la gran dependencia cultural y tecnológica de nuestro país, la mayor parte de los libros que contienen los temas tratados en estos apuntes no están escritos en nuestro idioma. Y aún las versiones inglesas de los libros considerados como fundamentales para una formación adecuada de los alumnos, no se encuentran disponibles en las bibliotecas y eso sin mencionar nada de las librerías del país. Tómese un libro clásico, por ejemplo *The mytical man* de Brooks el cual resulta imposible encontrarlo a ningún precio.

La dificultad para obtener dichos textos ha planteado la necesidad de escribir en nuestro idioma materiales que coadyuven al mejoramiento de la *calidad de la enseñanza* de la Máxima Casa de Estudios, tomando en cuenta las necesidades propias de nuestros educandos.

Estos apuntes se encuentran enfocados hacia los alumnos de segundo semestre de la carrera de ingeniería en computación. Tomando en cuenta que la sencillez con las que se han tratado los temas en su conjunto, logran hacer de estos apuntes, un material accesible al lector interesado en el área, aún cuando no se tengan grandes fundamentos al respecto.

IV.2 La filosofía de los apuntes de programación estructurada.

La elaboración de estos apuntes ha constituido un problema estructural que ha implicado una serie de tomas de decisión, que siempre se han tratado de orientar tomando como factor principal de decisión a los lectores hacia los que están dirigidos.

Tomemos en primer instancia los temas que marca el plan de estudios: La Programación estructurada (PE) y la Programación orientada a objetos (POO) son enfoques ortogonales de la misma materia. Tal y como se ha expuesto en el capítulo tres, ortogonales no significa contradictorios, sino formas distintas de ver una misma materia de estudio: **el modelado de la realidad por medio de una computadora.**

Por una parte, la programación estructurada representa uno de los enfoques más populares que existen actualmente en la construcción de programas. Su enfoque es claramente orientado a los procedimientos. Tiene una gran tradición dentro de los medios de desarrollo de software y en cierta medida ha dado lugar al desarrollo del Análisis estructurado y el Diseño estructurado. Su popularidad ha creado también una gran perversión de los conceptos que definen los límites de la Programación estructurada.

A esto hay que sumar el hecho de que la preparación de los profesionales en el área ha provocado que muchas personas se dediquen a construir sistemas sin tener una verdadera formación que los capacite para crear sistemas bien estructurados.

La programación estructurada ha sido muy malinterpretada a lo largo de todos estos años de crecimiento en esta área del saber humano.

Por otra parte, la Programación orientada a objetos, aun cuando sus orígenes se remontan casi a las mismas fechas que la Programación estructurada, es una metodología que ha comenzado a tomar un gran auge hasta años recientes. Las razones son por demás claras: se requiere de una estructura cognoscitiva mucho más solida para poder escribir programas, orientados a objetos, que funcionen.

La Programación orientada a los objetos no se encuentra orientada hacia los procedimientos, y no hace un gran énfasis en ellos. Parece ser una metodología que inicia de cero y que niega todos los principios empíricos bajo los cuales los programadores tradicionales han sido capacitados durante décadas.

No es de extrañar por tanto que parezcan metodologías disímbolas y excluyentes. Los mismos libros que apoyan estas filosofías enfatizan las diferencias al grado de expresar que *para aprender POO es necesario no saber nada de programacion estructurada* . La confusión que esto puede provocar en el alumno es muy grande. *¿ Cómo explicar que todo lo que se ha visto en el capítulo uno es inútil en el capítulo tres?.*

En las fases iniciales de la construcción de estos apuntes se pensó mucho en este problema. Y la solución más satisfactoria que se pudo encontrar fué fundamentar la programación orientada a objetos en base a los principios que posee en común con la programación estructurada. Esto implicó la revisión de algunos conceptos y la inclusión de algunos otros para dar integridad a la materia y que esta pudiera ser vista como un todo sin contradicciones.

Uno de los conceptos que resultó necesario redefinir fué el módulo. Debido a reflexiones personales respecto a lo que un módulo debe de ser y a la necesidad de utilizar este concepto para explicar lo que es un objeto, he propuesto la definición que aparece en estos apuntes. La importancia que reviste el concepto funcional de lo que un módulo *deber ser* y *deber hacer* puede fundamentar toda una teoría sobre la creación de software. Tanto la POO como la PE intentan explicar de manera funcional sobre lo que una unidad debe realizar dentro de un sistema. El punto común es un acoplamiento débil y una fuerte cohesión. Al ser estos conceptos fundamentales en ambos diseños no ha sido posible omitirlos.

La unidad de ambas filosofías ha sido algo que ha tratado de ser expuesto por distintos autores como Yourdon, Constantine o Barkakati. En estos apuntes se ha hecho todo lo posible por evitar la confusión que pudiese surgir en el alumno y darle una panorámica más amplia sobre lo que ambas filosofías son.

Desde mi punto de vista, el alumno debería de ser consciente de que existen múltiples formas de ver el desarrollo de sistemas y que ninguna de estas formas es, en sí misma, un fín sino un medio. Las filosofías existentes han sido el producto de una lenta y continua evolución de los lineamientos que son puestos en práctica, primero en forma empírica, y posteriormente como una metodología más formal. *Ninguna de estas filosofías está en sí misma acabada y ninguna pretende poseer la verdad.* Esto es muy importante repetirlo en las aulas una y otra vez.

La POO ha sido fundamentada en base a un pensamiento evolutivo que es explicado en el capítulo tres. El capítulo tres no puede ser cabalmente entendido si antes no se ha leído el capítulo uno. A su vez, la significancia de los conceptos del capítulo uno, solo puede ser cabalmente comprendida hasta que se ha terminado de estudiar el capítulo tres... o se han tenido muchos años de práctica en la programación estructurada y el diseño modular.

Como se puede apreciar esto constituye en sí mismo otro problema que se ha tratado de minimizar en la medida de lo posible. El hecho de que los lectores finales de estos apuntes sean escencialmente alumnos de segundo semestre me han colocado en una situación de difícil salida.

El nivel cognoscitivo de un alumno de segundo semestre no llega más allá de la construcción de algunos programas de unas cuantas líneas de código. El diseño de sistemas de grandes dimensiones o inclusive el diseño de sistemas de cualquier dimensión está fuera de su experiencia. Los problemas a los que cualquier analista, diseñador o programador se enfrenta son cosas totalmente desconocidas para un alumno. La importancia de los conceptos del capítulo uno pasan totalmente desapercibidos para alguien que no ve en ellos soluciones a problemas, sino temas obligatorios que estudiar.

La falta de tiempo disponible en el aula y el conocimiento poco profundo de un lenguaje de programación, impide al profesor de asignatura ejemplificar con ejemplos reales los conceptos que se vierten o las innumerables facetas y situaciones que se presentan en el desarrollo de un sistema de cómputo. Un ejemplo real es complejo en sí mismo, suele ser muy largo y requiere de un conjunto de conocimientos propios del área de aplicación del sistema. Conocimientos que el alumno generalmente tampoco posee. Esto lleva al profesor a presentar a sus alumnos ejemplos tan superficiales que no pueden reflejar la complejidad de un sistema real y por tanto dejan de cumplir con su misión ejemplificadora al crear un concepto erróneo sobre lo que un sistema es. Lo cual da por resultado que los alumnos sigan con una deficiencia en su visión de los sistemas, con la misma falta de experiencia y más lagunas conceptuales que cuando iniciaron.

La falta de experiencia del alumno genera un círculo vicioso

Esto se ha tomado en cuenta esto a lo largo de la escritura de los apuntes de tal manera que los ejemplos presentados se encuentren dentro del área cognocitiva de los alumnos, y en la suposición de que han cursado un semestre previo con un lenguaje de programación. Se ha procurado presentar ejemplos completos a lo largo de los capítulos, en donde ha sido necesario implantar en forma física las soluciones.

Tal ha sido la razón por la cual se ha explicado el uso de C++. Por experiencia he podido comprobar

que los conceptos en su forma puramente conceptual no son captados en toda su profundidad a menos que se acompañen de ejemplos prácticos. El estudio de la teoría de la programación orientada a objetos hace necesario el estudio de por lo menos un lenguaje orientado a los objetos.

Este enfoque no ha sido posible aplicarlo a la teoría de la programación estructurada. La experiencia con los grupos que he tenido a mi cargo como profesor me ha mostrado que los alumnos llegan con características y experiencias muy heterogéneas. En estos apuntes se parte de la idea de que el alumno sabe programar, pero no se exige que posea un profundo conocimiento, ni un lenguaje en particular. Los alumnos deben de aprender lenguaje PASCAL en el primer semestre. Pero por desgracia sus conocimientos en dicho lenguaje dejan mucho que desear. Esto ha acarreado una decisión más; didácticamente no es recomendable suponer que los alumnos saben lo que deben saber. En vez de hacer suposiciones que no llevan a ninguna parte, se ha incluido en forma más detallada de lo acostumbrado, las reglas para la escritura de pseudocódigo. Las reglas que propongo para su escritura, así como la manera de emplearlo en el diseño por refinamiento sucesivo, han probado ser prácticos en situaciones reales de trabajo, y lo que es más importante aún: el pseudocódigo ha resultado una herramienta didáctica muy poderosa.

El pseudocódigo no es un concepto realmente nuevo, pero las reglas formales propuestas son el producto de algunos años en mi ejercicio de la programación. Esta herramienta nos permite presentar el diseño de una manera independiente de cualquier lenguaje, de una forma clara, intuitiva y comprensible para cualquier alumno.

IV.3 El enfoque de los apuntes: Diseño - Programación

El temario de la materia es un muy genérico de tal forma que permite al profesor de asignatura tomar una serie de enfoques que pueden ser igualmente válidos.

Por un lado se tiene que la materia recibe el nombre de programación estructurada y el capítulo uno no incluye en forma explícita el tema de las *estructuras de control*, fundamento de la programación estructurada. Esto en realidad podría ser inadvertido si consideramos que se trata del temario de una materia llamada : *Programación estructurada y características de lenguaje* y cuyo nuevo temario ya excluyó el tema de *características de lenguaje*.

Lo que no es posible dejar de lado es que el temario se intema indiscriminadamente dentro del análisis, el diseño y la programación. Cruza una y otra vez los límites con la consiguiente libre-interpretación que cada profesor guste hacer de la materia. El temario permite enseñar Analisis Estructurado y es válido. O permite enseñar Diseño Estructurado y sigue siendo válido. Nuestro derecho a la libertad de cátedra así nos lo permite. Atendiendo exclusivamente a la materia, los temas paracen orientarse a un diseño puramente modular.

El refinamiento sucesivo es una técnica informal del diseño modular que no necesariamente tiene que estar ligado a la programación estructurada. El árbol de decisión y la tabla de decisión son herramientas muy útiles dentro de cualquier filosofía de programación y diseño. Son aplicables en ambas fases del desarrollo.

Observando con atención el temario puede observarse, sin embargo, que la línea a seguir va por los rumbos del diseño: Definición del problema, Identificación de los módulos, refinamiento sucesivo.

Pero es bien claro que una cosa es el Análisis Estructurado, otra el Diseño Estructurado, otra muy distinta el Diseño Modular y, por último, lo que debería ser el objeto de estudio de la materia: la Programación Estructurada.

¿Qué enfoque se podría tener? Los alumnos de segundo semestre pueden tener serios problemas para distinguir entre los límites de estas metodologías.

El capítulo tres sufre de los mismos males. Su enfoque parece estar claramente orientado al análisis... mezclado con el diseño y la programación: Definición del problema, Identificación de objetos y clases, determinación de los métodos, escritura del programa.

Este último capítulo planteó grandes problemas en la exposición. ¿Cómo puede plantearse la identificación de objetos y clases si el alumno no sabe lo que es un objeto y jamás ha visto la implantación física de uno de ellos? ¿Cómo explicar el funcionamiento de los métodos si su escritura es anterior a la escritura del programa principal ? ¿Cómo se puede mostrar de la manera más clara lo que el análisis o el diseño buscan, si nunca se han visto ejemplos programados?

Esta mezcla de los límites de las fases. Me ha llevado a concluir que el mejor enfoque, dado el nivel de los alumnos, es el de *diseño-programación* haciendo un mayor énfasis en la programación.

Por una parte tenemos la falta de experiencia de los alumnos en el diseño de cualquier sistema. Muy pocos alumnos tiene una idea clara de lo que es un sistema en forma real, cual es su arquitectura y cuales son los problemas reales a los que se enfrenta cualquier profesionista (sea al nivel que sea), cuando se enfrenta con ellos. Los conceptos no pueden ser dejados al aire y suponer que el alumno se dará una idea de lo que el concepto significa puede ser un craso error. Simplemente eso cae fuera de su experiencia. Es necesario proporcionarle algunos ejemplos completos en su forma real.

Por otra parte, esto implica hacer un gran énfasis en la implantación de los sistemas. Esto nos aleja de las labores de diseño de sistemas y esa tampoco es la idea del curso. Para poder escribir un programa, sea este bajo la filosofía que sea, es necesario primero un diseño del cual partir. Esto es posible y altamente deseable de hacerlo en el aula o en trabajos extraclase.

Pero el análisis es definitivamente una labor que requiere de alguna experiencia con los lenguajes o con los sistemas reales. Con riesgo de caer en un caso ridículo o fuera de marco, permítaseme tomar un ejemplo burdo: Si el alumno no supiera como construir una cúpula y se le hablase de las ecuaciones que permiten su diseño, pero jamás se le enseñara en forma física lo que es una cúpula, muy probablemente el alumno jamá s podrá construir una en su vida. Si a otro alumno se le proporcionan tabiques, cemento y la cimbra necesaria para construir una cúpula, pero no le enseñáramos los medios para planificarla, tal vez el alumno la pueda construir, después de muchos intentos fallidos. Pero sus métodos serán los menos adecuados. Por último, supongamos que un alumno nunca ha visto un edificio. Y se le proporciona toda la teoría para construir una torre con una enorme cúpula. Cuando el alumno tiene a su cargo a los albañiles para su construcción, estos pueden hacerle todas las trampas concebibles ya que el alumno nunca ha tenido contacto real con el proceso de construcción de una cúpula.

Los conceptos teóricos siempre deben ser acompañados con ejemplos reales. La supersimplificación de los ejemplos o la teorización pura de los conocimientos alejan al estudiante de la realidad que se desea dominar.

Los apuntes han sido elaborados pensando en proporcionar al alumno los conocimientos necesarios para diseñar programas y escribirlos posteriormente. El análisis no se ha aboradado en forma explícita ya que lo considero un tema avanzado, o para alumnos con alguna experiencia en el área. Esa es la razón por la cual no se abordan, ni el Análisis Estructurado ni el Análisis Orientado a Objetos.

Aún la parte correspondiente al diseño podría llegar a considerarse insuficiente. Sin embargo es la que mi experiencia como docente me ha demostrado que es adecuada para el nivel de nuestros estudiantes. Existen otras materias dentro de la currícula que deben cubrir con mayor amplitud los temas de diseño y análisis.

IV.4 Ingenieros, programadores y arte

Una de las falacias que con mayor amplitud se han repetido de generación en generación es: *Programar es un arte.* No estoy de acuerdo con semejante sofisma.

Educar a nuestros ingenieros con la idea de que programar es un arte, solo nos está llevando a un tremendo callejón sin salida en el que la Universidad debe evitar entrar.

Educar a una persona haciéndole creer que es un artista, trae curiosas repercusiones que se han visto tenazmente acentuadas en nuestro país , dada nuestra muy particular ideosincracia. Los estudios sobre las reacciones de los programadores-artistas se pueden encontrar en diversos títulos publicados en inglés y que en general coinciden en comentar en que el programador-artista es una persona que puede causar muchos problemas.

Jean-Dominique Warnier nos comenta atinadamente, que los ingenieros han sido educados para ser creadores. Que toda la vida se alienta su afán de inovar y su vanidad artista. Sin embargo, en la vida real, los ingenieros que realmente pueden conseguir empleos creativos son muy *pocos*. La gran mayoría de ellos se dedican a labores de mantenimiento. Está por demás demostrado que la labor de mantenimiento es febrilmente rechazada por los ingenieros noveles.

En el caso del software, la historia es la misma. Educamos ingenieros para crear nuevos sistemas. Pero no los formamos para dar mantenimiento a los sistemas ya existentes. Su vanidad de artista se ve muy afectada cuando se le asignan labores distintas a las creativas: documentar sistemas, depuración, etc.

Los programadores-artistas son muy reacios a aceptar críticas de su estilo de programación o de sus reglas en las cuales ellos se basan para construir sistemas. Casi siempre tienden a pensar que ellos son los poseedores absolutos de la verdad y no aceptan la posibilidad de que existan otros enfoques, igualmente válidos, para realizar su trabajo. Una persona con tan poco espíritu crítico está fuera de los ideales de libertad y amplitud de criterio que han sido el sueño de muchos universitarios notables.

La computación no nació ayer. Ha sido objeto de multitud de inovaciones y de estudios. Existen autores que ya son considerados como clásicos dentro de esta área. En los países del primer mundo se habla de una *Ciencia de la computación*. Nosotros aún hablamos del *Arte de la computación*.

La Universidad tiene como misión formar profesionistas capacitados. Pero la Nación no puede aún reconocer la diferencia entre un técnico y un ingeniero en computación. Muchos egresados salen de las aulas con la sensación de que han perdido muchos años y saben tanto de la materia como cuando entraron. Tienen la sensación de que la carrera consiste de aprender a manejar paquetes de computación o convertirse en usuarios finales de un sistema de información. No es de extrañarse entonces que exista una competencia entre los técnicos y los ingenieros. Competencia que no existe ni remotamente en ningun otra área de conocimiento. Un albañil jamás toma el lugar del arquitecto, ni una enfermera toma el lugar del cirujano.

Estos apuntes intentan en una forma muy modesta contribuir a subsanar esta atroz deficiencia. Se ha hecho mucho incapie en que existen diferentes tendencias, se han mencionado a los autores más relevantes dentro de nuestra área de estudio, se ha complementado el texto con numerosas citas, sin que estas llegen a agobiar al lector. En fin, se ha tratado de demostrar a todo lo largo de este texto, que la computación no es un arte, sino una ciencia que tiene sus pilares en una serie de pensadores y estudiosos que los alumnos deberían sentirse inclinados a conocer. Solamente cuando los alumnos se sientan ingenieros, la Nación dejará de verlos como técnicos altamente calificados.

IV.5 Profundidad de los temas

A pesar de esos deseos por elevar la calidad de la enseñanza y subsanar la falta de textos que la materia requiere, mil veces el espacio ha impedido ahondar en los temas.

El temario de la materia es terriblemente ambicioso. En diecisiete horas se deben cubrir los temas correspondientes a la programación estructurada, en otra diecisiete horas se debe cubrir la POO y en treinta horas el alumno debe manejar el lenguaje C.

Cada uno de los libros que fundamentan cada capítulo es una obra de extensión generosa. El tradicional libro de Yourdon *Techniques of Program Structure and Design* es una obra muy amplia de 523 páginas que trata sólo de programación estructurada. El libro de Constantine *Structured Design* es una obra de 473 páginas. El *Manual de referencia de C* de Schild consiste de 749 páginas. Como su nombre lo indica, esta obra es una manual de referencia y no enseña a programar. Otra obra clásica es *Object Oriented Design with Applications* de Grady Booch, la cual está escrita en 580 páginas. Esta obra solo trata el tema de diseño y no se aplica al análisis como lo hace *Designing Objet-Oriented Software* .Wirfs-Brock R., Wilkerson B. y Wiener L.de 314 páginas Y ninguno de ellos enseña a programar. Para ello se debe recurrir a Eckel Bruce: *Aplique C++* de 521 páginas.

Profundizar en los temas requeriría de la escritura de otras dos obras de igual extensión a la que se tiene en estos momentos entre las manos. Tómese el caso de C y C++. Sería muy deseable mostrar el uso de esos lenguajes para sistemas con interfases Gráficas de Usuario. Windows de Microsft, está escrito en C++ y actualmente tiene una gran demanda. Los alumnos podrían aprender las técnicas de la POO en un ambiente desarrollado bajo POO como lo es Windows. Como efecto secundario de esto, sus aplicaciones lucirían con el profesionalismo esperado de los alumnos de una Escuela Nacional. Se insiste, se han sacrificado estos indudables beneficios en pro de la sencillez, la portabilidad del código, la consideración de la falta de experiencia de los alumnos y la terrible escases de equipo apropiado en la ENEP. Equipo que no se encuentra disponible ni para aprender C++.

Cuando se están leyendo los apuntes, se tiene siempre la sensación de que se puede decir más, de que se puede profundizar más, de que se puede dar una explicación más amplia. Pero los límites de la obra que debe ser publicada y el costo para los alumnos impide extender los temas libremente. Se han escrito unos apuntes de programación que sean una referencia, para que el profesor pueda ocuparse más en la práctica de los conceptos dentro del aula y a la ampliación, y refinamiento de los temas. Los apuntes pueden ser consignados a los alumnos como lecturas extraclase.

La omnipresente falta de tiempo se encuentra en cada curso de programación. En este curso de diseño y análisis se resiente aún más. Los alumnos deben tomar consciencia de muchas cosas en un sólo semestre. Sería deseable que diseñaran por lo menos un sistema estructurado y un sistema orientado a objetos a lo largo del curso. Sería deseable el establecimiento de un laboratorio de programación obligatorio.

El alumno debe aprender los temas programando. Así como no se aprende a conducir un auto sin salir a manejalo, no se puede aprender a construir sistemas si no se programan sistemas.

Estos apuntes han sido escritos tratando de cuidar todos los aspectos mencionados y pensando siempre en los alumnos a quienes va dirigido.

Formar a profesionistas capacitados es una labor escencial de la Universidad y, la Nación así lo está demandando.

Marcas Registradas

IBM™ PC™	International Business Machines,Corp
MS-DOS™	Microsoft Corporation
Turbo C™	Borland International, Inc
UNIX™	AT & T
WINDOWS™	Microsoft Corporation
Clipper™	Nantucket Corporation
Dbase™	Borland International, Inc
Lotus™	Lotus Incorporated

APÉNDICE A

Sintaxis del lenguaje C

A.1 DECLARACIONES DE TIPO

Una **declaración de tipo** es la definición del tipo de datos que va a contener una variable. Dicha declaración tiene la forma general:

tipo: lista_de_variables
Donde

tipo es un tipo de datos válido en C
lista_de_variables consiste de uno o más identificadores separados por comas

Los rangos mínimos estipulados por la comisión ANSI son:

TIPO	TAMAÑO EN BITS	RANGO MINIMO
int	16	-32 767 a 32 767
unsigned int	16	0 a 65 535
signed int	16	-32 767 a 32 767
short int	16	-32 767 a 32 767
unsigned short int	8	0 a 65 535
signed short int	8	-32 767 a 32 767
long int	32	-2 147 483 647 a 2 147 783 647
signed long int	32	-2 147 483 647 a 2 147 783 647
unsigned long int	32	0 a 4 294 967 295
float	32	Seis dígitos de precisión
double	64	Diez dígitos de precisión
long double	128	Diez dígitos de precisión

A.2 EL OPERADOR DE ASIGNACIÓN

El operador de asignación puede aparacer en cualquier expresión válida. Para realizar una asignación no es necesario dedicar una línea especialemente para el efecto. La asignación puede realizarse en forma anidada dentro de otras estructuras de control. La forma general de una asignación es:

variable = expresión;

Donde

variable es un nombre de variable o un puntero válido
expresión es una operación tan compleja como sea necesario o una simple constante

A.3 printf()

La forma general de la salida por formato es la siguiente.

#include <stdio.h>
int printf(cadena de control, lista de argumentos)

Donde:

cadena de control	Es una cadena con los códigos que controlarán la forma como de desplegarán los resultados en el dispositivo de salida
lista de argumentos	Es la lista con las variables o las expresiones que serán desplegadas.

La función devuelve el número de caracteres escritos. Un valor negativo indica error en la escritura.

Los códigos para formato se encuentran listados en la siguiente tabla

Código	Formato
%c	Caractér
%d	Enteros decimales con signo
%i	Enteros decimales con signo
%e	Notación cientifica (e minúscula)
%E	Notación cientifica (E mayúscula)
%f	Coma flotante
%g	Usar %e o %f, el que resulte más corto
%G	Usar %E o %F, el que resulte 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	Mostrar puntero
%n	El argumento asociado es un puntero a entero al que se asigna el número de caracteres escritos
%%	Imprimir el signo %

Codigos de control

La siguiente es una tabla de los tripletes aceptados.

Código	Significado
\b	Espacio atrás
\f	Salto de página
\n	Salto de línea
\r	Salto de carro
\t	Tabulador
\"	Comillas
\'	Apóstrofo
\0	Nulo
\\	Barra invertida
\v	Tabulador vertical
\a	Alerta (Sonido)
\N	Constante octal
\xN	Constante hexadecimal

Código de barras invertidas

A.4 scanf()

La forma general de la función `scanf()` es muy parecida a la de `printf()` y se muestra a continuación:

#include <stdio.h>
int scanf (cadena de contol, lista de argumentos)

Donde:

cadena de control	Es una cadena con los códigos que controlarán la forma como se recibirán los datos desde el dispositivo de entrada.
lista de argumentos	Es la lista con las direcciones de las variables que serán leídas.

La cadena de control le indica a **scanf()** que tipo de datos debe esperar para poder representarlos internamente de la manera correcta. La función devuelve el número de variables que han sido leídas exitósamente. Devuelve cero si no se realizó,es decir, si no se asignó ningún valor a la lista de variables.

La siguiente es una tabla de los especificadores de **scanf()**

Código	Formato
%c	Caractér
%d	Enteros decimales con signo
%i	Enteros decimales con signo
%e	Notación científica (e minúscula)
%f	Coma flotante
%g	Usar %e o %f, el que resulte más corto
%o	Octal sin signo
%s	Cadena de caracteres
%x	Hexadecimales sin signo (letras minúsculas)
%p	Mostrar puntero
%n	Recibe un valor entero igual al número de caracteres leídos
%u	Lee un entero sin signo
%[ ]	Muestra un conjunto de caracteres

Códigos de control para scanf()

A.5 SENTENCIA FOR

La iteración FOR se ejecuta cero o más veces dependiendo de una expresión lógica. La sintaxis es la siguiente:

for (expresion1; expresion2;expresion3) sentencia;

Donde:

expresion1 : es la expresión evaluada al inicio del ciclo
expresion2 : es la expresion que detendrá al ciclo cuando esta evalúa en cero
expresión3 : es una expresion que será evaluada cada vez que se ejecute un nuevo ciclo.
sentencia : es la sentencia que conforma el cuerpo del for y que se desea ejecutar.

A.6 ARREGLOS

La forma general para declarar un arreglo es:

tipo nombre_de_variable[número_de_elementos]

La forma general para declarar un arreglo multidimensional es:

tipo nombre_del_arreglo[d1][d2][d3][d4]...[di]...[dn]
Donde
d es la longitud del arreglo en la i-ésima dimensión
n es la cantidad de dimensiones que contiene el arreglo

A.7 SENTENCIA WHILE

La forma general del while es:

```
while (condición)
        sentencia;
```

Donde:

condición: es una expresión relacional o lógica.
sentencia : es cualquier sentencia válida en C.

La forma general de un ciclo while para un bloque de instrucciones es la siguiente:

```
while (condicion)
{
        primera sentencia;
        segunda sentencia;
        .
        .
        .
        ultima sentencia;
}
```

A.8 SENTENCIA IF

La forma general de una sentencia if es:

```
if (condición)
        sentencia que se ejecuta si la condición es verdadera;
else
        sentencia que se ejecuta si la condición es falsa;
```

O bien en su forma de bloques:

```
if ( condición)
{
        primer sentencia del bloque verdadero;
        segunda sentencia;
        .
        .
        .
        ultima sentencia del bloque verdadero;
}
else
{
        primera sentencia del bloque falso;
        segunda sentencia;
        .
        .
        .
        ultima sentencia del bloque falso;
}
```

Los operadores para una condición, así como su precedencia se encuentran resumidos en las siguientes tablas:

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

Tabla de operadores

!	mayor
> >= < <=	
== !=	
&&	
||	menor

Tabla de precedencia

A.9 SENTENCIA SWITCH

La estructura de decisión múltiple a través de la sentencia switch cuya forma general es :

```
switch(expresión)
{
        case constante1:
                bloque 1 de sentencias;
                break;
        case constante2:
                bloque 2 de sentencias;
                break;
        case constante3:
                bloque 3 de sentencias;
                break;
        .
        .
        .
        case constanten:
                bloque n de sentencias;
                break;
        default:
                bloque n+1 de sentencias;

}
```

A.10 SENTENCIA DO-WHILE

La forma general de un *do - while* es

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

o su forma en bloque:

```
do
{
        primera sentencia;
        segunda sentencia;
        .
        .
        .
        ultima sentencia;
}       while (condicion);
```

A.11 FUNCIONES PROTOTIPO

La forma general de un prototipo de función es el siguiente:

```
tipo nombre_de_funcion( lista de parámetros)
```

A.12 DECLARACIÓN DE PARÁMETROS

Existen dos formas de realizar la declaración de los parámetros de la función. La más antigua es:

```
nombre_de _funcion (lista de parámetros)
declaración de parámetros
{
        bloque de instrucciones
}
```

Esta forma es conocida como la *forma clásica* definida por Ritchie & Kernigan. El estandar ANSII define una forma alterna para la declaración que es conocida como la *forma moderna*. La forma moderna es definida de la siguiente manera:

```
tipo nombre_de_funcion (tipo argumento_1, tipo argumento_2, tipo argumento_3,...,tipo argumento_n)
{
        bloque de instrucciones
}
```

A.13 DIRECTIVAS #ifdef, #ifndef, #endif

Las directivas `#ifdef y #ifndef` inquieren por la definicion de una macro, su forma general es:

```
#ifdef nombre_de_macro
        bloque de sentencias
#endif
```

Si se ha definido la macro especificada entonces se ejecuta el bloque de sentencias englobado hasta el `#endif`.

O bien su contraparte :

```
#ifndef nombre_de_macro
        bloque de sentencias
#endif
```

A.14 DIRECTIVA `#error`

Esta directiva fuerza al compilador a desplegar un mensaje de error en la pantalla para posteriormente detener la compilación. El mensaje no debe ir entre comillas. . La forma general es:

```
#error mensaje_de_error
```

Esta directiva es sumante útil en el proceso de depuración.

A.15 DIRECTIVAS `#if, #elif, #else y #endif`

La forma general para usar una directiva condicional en el preprocesador de C es:

```
#if expresion-constante
        bloque verdadero de sentencias
#else
        bloque falso de sentencias
#endif
```

La directiva #elif suministra la selección múltiple. Es equivalente a realizar un anidamiento de decisiones. Su forma general es:

```
#if expresion
        bloque de sentencias
#elif
        bloque de sentencias
#elif
        bloque de sentencias
.
.
.
#endif
```

A.16 DIRECTIVA #include

La forma general de la directiva #include es:

#include <archivo>

O su forma alterna:

#include "ruta_de_acceso_archivo"

A.17 ESTRUCTURAS

La declaración de una estrcutura se realiza por medio de la sentencia **struct**, su forma general es:

```
struct nombre_estructura
{
        tipo identificador ;
        tipo identificador ;
        tipo identificador ;
        .
        .
        .
        tipo identificador ;
} lista de variables_de_estructura;
```

Donde:

nombre_estructura: Representa el nombre que recibirá la estructura. Dicho nombre será único para la estructura.

identificador: Es el nombre de un componente de la estructura, cada componente puede ser de tipo elemental o compuesto.

variable_de_estructura: Es el nombre de una variable que estará conformada por los tipos declarados en la estructura.

Se puede omitir el *nombre_estructura* o bien la *lista de variables de estructura* pero no ambas.

La forma general para definir una estructura cuyos miembros accesan directamente a los bits es la siguiente:

```
struct etiqueta
{
        tipo identificador: longitud ;
        tipo identificador: longitud;
        tipo identificador: longitud;
        .
        .
        .
}lista_de_variables;
```

Donde:

tipo	Puede ser int, unsigned o signed.
longitud	Es el número de bits que conformarán el campo.

Los campos de un solo bits deben declarase como unsigned ya que un bit no puede tener signo.

A.18 UNIONES

La forma general para realizar una union es la siguiente

```
union nombre_union
{
        tipo  identificador ;
        tipo identificador ;
        tipo identificador ;
        .
        .
        .
        tipo identificador ;
} lista_de_variables;
```

A.19 ASIGNACIÓN DINÁMICA DE MEMORIA

La función `malloc()` (memory allocation = localización de memoria) realiza una petición de memoria
La función prototipo de la `malloc()` es la siguiente:

```
#include < stdlib.h>
void *malloc( size_t número_de_bytes );
```

Donde:

size_t	Se encuentra definida en *stdlib.h* como un entero sin signo
numero_de_bytes	Es la cantidad en bytes que deseamos obtener de memoria

La contraparte de la función `malloc()` es la funcion `free()` (free= libre) la cual libera la memoria.
Su forma general es

```
#include <stdlib.h>
void free ( void *puntero)
```

A.20 APERTURA DE ARCHIVOS : fopen(), fclose()

La función que realiza la apertura de un archivo es **fopen()** y tiene la siguiente forma general:

```
#include <stdio.h"
FILE *fopen(const char *nombre_archivo, const char *modo) ;
```

Donde:

FILE	Es un tipo especificado en en el archivo de cabecera stdio.h
nombre_archivo	Es el nombre del archivo que se debe abrir
modo	Es alguno de los modos válidos.

Si la función tiene existo devuelve un puntero a un archivo, de lo contrario retoma un valor nulo.

Los modos de apertura son listados a continuación:

a) Modo r :	En este modo se abre un archivo para ser solo para lectura. El archivo debe exitir previamente
b) Modo w:	Un archivo es creado para ser utilizado solo en escritura, si el archivo existe su contenido es destruido
c) Modo a:	Un archivo es abierto para añadir al final del mismo.
d) Modo rb:	Abre un archivo binario para solo lectura
e) Modo wb	Crea un archivo binario para escritura
f) Modo ab	Añade información a un archivo binario
g) Modo r+	Abre un archivo existente para operaciones de lectura/escritura
h) Modo w+	Crea un archivo para operaciones de lectura/escritura
I) Modo a+:	Añade o crea un archiva para operaciones le lectura/escritura
j) Modo rb+:	Abre un archivo binario para operaciones de lectura/escritura
k) Modo wb+:	Crea un archivo binario para operaciones de lectura/escritura
l) Modo ab+:	Añade información a un archivo binario para operaciones de lectura/escritura

Modos de apertura de un archivo

Cuando un archivo ya no será necesitado por un programa, es muy recomendable que el archivo sea *cerrado*, es decir que se elimine la conexión entre el programa y el archivo. La función **fclose()** cumple con dicha tarea. Su función prototipo es la siguiente:

Int fclose(FILE *puntero_archivo);

Donde:

puntero_archivo Corresponde al puntero del archivo que se desea cerrar

Si la función tiene éxito se devuelve un valor de cero, en cualquier otro caso se devuelve un valor distinto de cero.

A.22 LECTURA/ESCRITURA EN UN ARCHIVO

putc(), fputc()

La función **putc()** y la funcion **fputc()** son equivalentes. Con ellas se puede escribir un caractér en un archivo. Su función prototipo es:

#include <stdio.h>
Int fputc(Int caracter, FILE *puntero_archivo)

Donde:

caracter Es el caractér que se va a escribir en el archivo.
puntero_archivo Es el puntero a un archivo abierto

Si la función produce un error devuelve EOF. El valor devueltoo por la función es el valor del caracter escrito.

fget(), getc(), foef()

La lectura de un archivo se realiza con la función **fgetc()** y **getc()** que son idénticas. La forma general de dicha función es:

#Include <stdio.h>
int fgetc(FILE *puntero_archivo);

Devuelve el valor del siguiente caracter en el archivo de entrada e incrementa el valor de la posición del archvo. Si se alcanza fin de archivo se retorna EOF.

Una segunda función que es importante para realizar la lectura de una archivo es **feof()**, la cual inquiere por el fin del archivo. Su función prototipo es:

#Include <stdio.h>
int feof(FILE *puntero_archivo);

En caso de que nc se alcance el fin de archivo la función devuelve cero. Cuando se llega al final del archivo se obtiene el valor de cierto

fputs(), fgets()

Trabajar con caracteres individuales puede resultar tedioso, en su lugar se leen cadenas completas a través de las funciones **fputs()** y **fgets()**.

La función prototipo para **fputs()** es la siguiente:

int fputs(const char *cadena, FILE *puntero_archivo)

Donde:

**cadena* Es el puntero de la cadena que se va a ecribir en el archivo
**puntero_archivo* Corresponde al archivo en el cual se desea escribir

El caracter de fin de cadena no es escrito en el archivo.. La función devuelve un valor negativo cuando ha tenido éxito, en cualquier otro caso devuelve EOF.

La función prototipo para **fgets()** es la siguiente:

int *fgets(char *cadena, int longitud_de_cadena, FILE *puntero_archivo)

Donde:

**cadena* Es el puntero de la cadena en donde se almacenarán los caracteres leídos en el archivo

longitud_de_cadena Es el número de caracteres que se leerán como parte de la cadena.

**puntero_archivo* Corresponde al archivo del cual se leerá la información

La función leerá longitud_de_cadena-1 caracteres que se almacenarán en la cadena. Si se encuentra un salto de línea, se toma como fin de cadena. Si cualquiera de las funciones finaliza con error devolverán un valor de EOF.

fwrite(), fread()

Un archivo de datos suele tener más de un tipo de dato. Para realizar una escritura a un archivo con datos que ocupan más de un byte se hace uso de la función **fwrite()** la cual tiene la siguiente función prototipo:

size_t fwrite(void *buffer, size_t número_de_bytes, size_t cuenta, FILE *puntero_archivo)

Donde:

size_t Es un tipo definido en stdio.h y es semejante al tipo unsigned

buffer Es un puntero a la localidad de memoria de donde serán escritos los datos hacia el archivo.

numero_de_bytes Número de bytes que se escribirán en el archivo

cuenta Es el número de elementos con *numero_de_bytes* de longitud que se escribirán en el archivo

puntero_archivo Corresponde al archivo en el cual se grabará la información.

La función retoma el número de elementos escritos en el archivo. Si la función termino en forma correcta, entonces dicho valor será igual al de *cuenta*.

La contraparte de `fwrite()` es `fread()`. Su función prototipo es la siguiente:

size_t fread(void *buffer, size_t número_de_bytes, size_t cuenta, FILE *puntero_archivo)

Donde:

size_t Es un tipo definido en stdio.h y es semejante al tipo unsigned

buffer Es un puntero a la localidad de memoria a donde serán escritos los datos procedentes del archivo.

numero_de_bytes Número de bytes que se leerán del archivo

cuenta Es el número de elementos con *numero_de_bytes* de longitud que se leerán del archivo

puntero_archivo Corresponde al archivo del cual se leerá la información.

fseek()

En lenguaje C el acceso a una registro directo se logra a través de la función **fseek()**, la cual tiene la siguiente función prototipo:

int fseek (FILE , *puntero_archivo, long numero_bytes, int origen)

Donde:

puntero_archivo Corresponde al archivo en el cual se realizará una operación de lectura/escritura

numero_bytes Es la cantidad de bytes que se movera el puntero del archivo con respecto a la dirección de inicio de archivo.

origen Indica el origen del desplazamiento del puntero que se puede realizar a partir del principio del archivo, de la posición actual o de el fin del archivo

La especificación del oigen se puede realizar a través de las siguientes macros definidas en **stdio.h**

SEEK_SET	Principio del archivo
SEEK_CUR	Posición actual
SEEK_END	Fin del archivo

La función **fseek()** retoma cero si se concluyó la tarea sin errores.

fprintf(), fscanf()

Básicamente **fprintf()** y **fscanf()** funcionan de la misma manera que lo hace **printf()** y **scanf()** respectivamente. Sus funciones prototipo son:

int fprintf(FILE *puntero_archivo, const char *cadena_de_control, lista_de_variables);

int fscanf(FILE *puntero_archivo, const char *cadena_de_control, lista_de_variables);

APÉNDICE B

Secuencias ANSI para el control de la pantalla

Las secuencias ANSI pueden utilizarse en cualquier terminal de PC bajo DOS que sea arrancada con el archivo config.sys conteniendo la linea:

device=ansi.sys

O bien en las terminales VT 100 para los sistemas UNIX. Existen muchos terminales que disponen de una emulación para este tipo de terminal.

Recuerde al utilizar estas secuencias, que se hace uso de un manejador de pantalla que retardará un poco la ejecución de sus programas y que obliga al usuario a disponer del manejador respectivo dentro de su ambiente de trabajo.

SECUENCIA	ACCION
\x1b[renglonesA	Mueve el cursor n renglones hacia arriba
\x1b[renglonesB	Mueve el cursor n renglones hacia abajo
\x1b[columnasC	Mueve el cursor n columnas a la derecha
\x1b[columnasD	Mueve el cursor n columnas a la izquierda
\x1b[renglon,columnaH	Mueve el cursor a una posición absoluta en la pantalla
\x1b[2J	Borra pantalla
\x1b[K	Borra la linea
\x1b[s	Salva la posición de cursor
\x1b[u	Restablece la posición del cursor
\x1b[0m	Video normal
\x1b[1m	Video sobresaltado
\x1b[5m	Video parpadeante
\x1b[7m	Video inverso
\x1b[8m	Invisible
\x1b[30m	Negro
\x1b[31m	Rojo
\x1b[32m	Verde
\x1b[34m	Azul
\x1b[35m	Marrón
\x1b[36m	Cian
\x1b[37m	Gris

Ejemplos:

```
void Borrapantalla(void)
{
        printf ("\x1b[2J");
}

void Gotoxy(int col,int ren)
{
        printf("\x1b[%d,%dH", ren,col );
}
void pon_inverso(void)
{
        printf("\x1b[7m");
}
```

APÉNDICE C

Glosario

abstracción: La característica escencial de un objeto que lo distinge de todos los otros objetos y que provee una definición conceptual de los límites realativos a la perspectiva del observador; el proceso de enfocar las características escenciales de un objeto. La abstracción es un elemento fundamental del diseño orientado a los objetos.

acoplamiento: Fuerza de unión entre módulos de un sistema

algortimo: Especificación detallada y libre de ambigüedad paso a paso de la solución de un problema

analisis: Etapa de desarrollo de sistemas en la cual los analistas y los usuarios establecen las especificaciones del sistema que se desea implantar. Se clarifican los objetivos dentro del dominio del lenguaje del problema.

analisis estructurado: Un conjunto de principios y técnicas que asisten al análista de sistemas en el establecimiento de los requerimientos funcionales de un sistema en términos lógicos.

analisis orientado a objetos: Un método de análisis en el cual los requerimientos son examinados desde la perspectiva de las clases y los objetos encontrados en el vocabulario que es dominio del problema.

análisis procedimental: Un conjunto de criterios que determinan que módulos deben ser conjuntados en una sola unidad para lograr una mejor eficiencia;

analista de sistemas: Persona que define un problema, determina con exactitud lo que se requiere para resolverlo y define el formato general de la solución por computadora.

ANSI: Abreviatura del American National Standards Institute.

apuntador: Vease: puntero

árbol de decisión: Herramienta de diseño que consiste en un diagrama que representa las diversas condiciones que pueden encontrarse dentro de un sistema de toma de decisiones, y que afectan de manera significativa en comportamiento del mismo

arquitectura de espagetti: Dicese de aquellos sistemas con un exceso de saltos incondicionales. En general se dice de aquellos sistemas con estructuras de control inestructuradas o inexsistentes.

ASCII: Abreviatura de American Standard Code for Information Interchange

bottom-up: Vease: diseño ascendente.

caida del sistema: Se conoce así al fenomeno que ocurre cuando un sistema sufre un error irrecuperable y el operador pierde el control sobre la computadora.

clase: Un conjunto de objetos que comparten una estructura común y una conducta común. Los términos de clase y tipo suelen (pero no simpre) ser intercambiables; una clase es ligeramente diferente del concepto de tipo, en que se esfatiza la importancia de la jerarquia de clases.

clase abstracta: Una clase que no posee instancias. Una clase abstracta es escrita esperando que sus subclases añadan su estructura y su conducta, usualmente completando la implantación de sus métodos típicamente incompletos.

clase base: La más generalizada clase en una estructura de clases. La mayoria de las aplicaciones poseen tales clases bases.

cliente: Un objeto que usa los recursos de otro, sea por su operación o por referencia a su estado.

cohesión: Grado de relación funcional de los elementos de procesamiento dentro de un módulo sencillo.

compilador: Programa que sirve para traducir un lenguaje de programa simbólico más alto a un lenguaje de máquina que sea comprensible para el procesador. La entrada a un compilador es un programa fuente. El compilador asigna asigna partes de la memoria interna a los datos necesarios para el programa; lleva un registro temporal de nombres, localizaciones y características de cada elemento dato en una lista denominada tabla simbólica; interpreta especificaciones de procedimiento del programa fuente, y determina que instrucciones de la computadora son necesarias para realizar el trabajo. Después el compilador genera las instrucciones de máquina; les asigna almacenamiento; proporciona al programa las direcciones correctas de datos y las constantes requeridas; y produce así un programa ejecutable.

computabilidad: La teoría de la computabilidad establece que un proceso puede ser descrito mediante un modelo matemático construido con un lenguaje simbólico sencillo, y que existen métodos formales para reproducirlo por medio de un autómata.

Más aún, un resultado central de esta teoría descubre la existencia de ciertos procesos que no pueden ser reproducidos ni siquiera por medio de ningún autómata y que por tanto, están más allá de lo que forma el universo de conceptos cognocibles en forma algotítmica por el ser humano.

conducta: Como un objeto actua o reacciona, en términos de sus cambios de estado y envio de mensajes.

confiabilidad: Ausencia de falla expresada por lo general como probabilidad que una falla no ocurra durante cierta duración de uso. La confiabilidad depende de la repetibilidad y se expresa como un nivel de confianza.

constructor: Una opración que crea un objeto y/ o inicializa su estado.

depuracion: Proceso de identificación y eliminación de errores, fallas de funcionamiento o fallas de un programa

destructor: Una operación que libera el estado de un objeto y/o lo destruye a él mismo.

diagrama de flujo: Método gráfico para describir algoritmos.

dirección: Número de identificación para un registro, localización de memoria o estación en una red de comunicaciones. Definido en términos generales, es cualquier parte de una instrucción que especifique la ubicación de su operando (aquello que es operado por la instrucción). Cumple con la misma función de hallar la información que las direcciones de una calle para encontrar una casa

diseño: Fase del desarrollo de sistemas en la cual se establece la arquitectura del sistema. Primero se establece la forma lógica en la cual se interrelacionarán los elementos, la segunda fase consiste del desarrollo de la forma física en la cual se implantará el sistema (módulos, archivos, bases de datos, etc)

diseño ascendente: Método que consiste en partir de problemas particulares e ir encontrando sus características comunes para elaborar las rutinas más generales que controlarán el sistema.

diseño descendente: Método utilizado en la programación estructurada. Programar por refinamiento sucesivo. La tarea se descompone repetidamente en subtareas hasta que cada subproblema pueda resolverse.

diseño estructurado: Un conjunto de principios y técnicas que asisten al diseñador de sistemas en la determinación de la interconexión de los módulos y la mejor manera de resolver un problema.

diseño estructural: El diseño de la estrcutura de un sistema: la especificación de las piezas (v. gr. módulos) y la interconexión entre las piezas.

diseño orientado a los datos: Un tipo de estrategia de diseño que deriva de un diseño estrcutural tomando en consideración las estructuras de los conjuntos de datos asociados con el problema.

diseño orientado a objetos: Un método de diseño que redunda en el proceso de la descomposición orientada al objeto y a una notación lógica y física así como modelos estáticos y dinámicos del sistema bajo diseño; especificamente, esta notación incluye diagramas de clase, diagramas de objeto, diagramas de módulo y diagramas de proceso.

dispositivo: Componente de computadora o la computadora misma. En un sentido más amplio, aparato o mecanismo que es creado, formado, inventado o construido por diseño.

encapsulamiento: Vease: ocultiamiento de información

enlace dinámico: Enlace denota la sociación de un nombre (tal como una declaración de variable) con una clase; el enlace dinámico es un aenlace en el cual la asociación nombre/clase no es hecha hasta que el objeto es designado por el nombre con el cual fué creado en tiempo de ejecución.

enlace estático:Enlace denota la sociación de un nombre (tal como una declaración de variable) con una clase; el enlace estático es un enlace en el cual la asociación nombre/clase es hecha cuando el nombre es declarado (en tiempo de compilación) pero antes de la creación del objeto al cual designa con un nombre.

ensamblador: En su primera acepción, un ensamblador es un programa que recibe programas escritos en un lenguaje (que también se llama ensamblador) y los traduce a lenguaje de máquina.

La segunda acepción se refiere al lenguaje que reconoce un programa ensamblador, y que constituye el segundo nivel de complejidad de los lenguajes con que se puede programar una computadora.

estado: Una de las posibles condiciones en la cual un objeto puede existir, caracterizado por la definición de cantidades que se distingen de otras cantidades; en cualquier punto en el tiempo, el estado de un objeto resulta en todas las propiedades del objeto (sualmente estáticas) más los valores de esas propiedades (usualmente dinámicas)

estructura: La representación concreta del estado de una entidad.

estructura de control: La estructura de un programa definida por referencias con las cuales se representan las transferencias de control. Construcciones mediantes las cuales se escriben los programas.

estructuras de datos: Métodos que se emplean en programación para organizar y representar la información en una computadora.

función: Relación que define el valor de una variable dependiente (Y) basada en el valor de una variable independiente (X); por ejemplo, una ecuación tal como Y = f(X). En el contexto de un analisis de requerimiento orientado a objetos, una conducta visible y probable.

función miembro: Una operación dentro de un objeto, definida como parte de la declaración de clase; todas las funciones miembro son operaciones, pero no todas las operaciones son funciones miembro. Los términos de función miembro y método son intercambiables.

función virtual: Una operación dentro de un objeto. Una función virtual puede ser redefinida por las subclases; esto es, para un objeto dado, su implantación a través de métodos en varias clases que están relacionadas por jerarquía de herencia.

funcion virtual pura: Aquella función virtual que no debe tener implantación dentro de la clase base que la define.

hardware: Los circuitos electrónicos y dispositivos electromecánicos que constituyen el sistema de computación. Cualesquiera partes físicas del sistema incluyendo circuitos integrados, terminales de video, impresora, mandos de juego, y dispositivos auxiliares de memoria.

herencia: Una relación entre clases en donde una clase comparte la estructura y la conducta definida en una (herencia simple) o varias (herencia múltiple) clases. La herencia define jerarquía del tipo "una clase de" entre clases en la cual una subclase hereda de una o más superclases; una subclase tipicamente aumenta o redefine la estrcutura y la conducta existente en su superclase.

identidad: La naturaleza de un objeto que lo distingue de todos los demás objetos.

identificador: El nombre, dirección, etiqueta o índice distintivo de un objeto en un programa.

implantación: En la POO, vista interior de un objeto, clase o módulo, incluyendo las caracteristicas elementales (secretas) de su conducta.

indecible: (problema) Dícese de aquellos problemas que no poseen una solución algoritmica

independencia: Término usado para describir pares de módulos: Se dice que dos módulos son independientes si cada unp puede funcionar completamente sin la presencia del otro.

instancias: Un ejemplo de clase. Una instancia posee estado, conducta e identidad. La estructura y la conducta de instancias similares están definidas en su clase común. Los términos de instancia y objeto son intercambiables

insumos: Los datos de los cuales se alimenta un sistema

integración de sistemas: Fase del desarrollo tradicional de sistemas. Cuando ya se poseen codificados todos los módulos del sistema, o todos los subsistemas (los cuales han sido desarrollados en forma paralela por distintos programadores) se procede a unirlos en un solo sistema. A esta etapa de unión se le conoce como integración de sistemas y culmina cuando todos los subsistemas interactúan correctamente.

enlace estático: Enlace denota la sociación de un nombre (tal como una declaración de variable) con una clase; el enlace estático es un enlace en el cual la asociación nombre/clase es hecha ciuando el nombre es declarado (en tiempo de compilación) pero antes de la creación del objeto al cual designa con un nombre.

ensamblador: En su primera acepción, un ensamblador es un programa que recibe programas escritos en un lenguaje (que también se llama ensamblador) y los traduce a lenguaje de máquina.

La segunda acepción se refiere al lenguaje que reconoce un programa ensamblador, y que constituye el segundo nivel de complejidad de los lenguajes con que se puede programar una computadora.

estado: Una de las posibles condiciones en la cual un objeto puede existir, caracterizado por la definición de cantidades que se distingen de otras cantidades; en cualquier punto en el tiempo, el estado de un objeto resulta en todas las propiedades del objeto (sualmente estáticas) más los valores de esas propiedades (usualmente dinámicas)

estructura: La representación concreta del estado de una entidad.

estructura de control: La estructura de un programa definida por referencias con las cuales se representan las transferencias de control. Construcciones mediantes las cuales se escriben los programas.

estructuras de datos: Métodos que se emplean en programación para organizar y representar la información en una computadora.

función: Relación que define el valor de una variable dependiente (Y) basada en el valor de una variable independiente (X); por ejemplo, una ecuación tal como Y = f(X). En el contexto de un análisis de requerimiento orientado a objetos, una conducta visible y probable.

función miembro: Una operación dentro de un objeto, definida como parte de la declaración de clase; todas las funciones miembro son operaciones, pero no todas las operaciones son funciones miembro. Los términos de función miembro y método son intercambiables.

función virtual: Una operación dentro de un objeto. Una función virtual puede ser redefinida por las subclases; esto es, para un objeto dado, su implantación a través de métodos en varias clases que están relacionadas por jerarquía de herencia.

funcion virtual pura: Aquella función virtual que no debe tener implantación dentro de la clase base que la define.

hardware: Los circuitos electrónicos y dispositivos electromecánicos que constituyen el sistema de computación. Cualesquiera partes físicas del sistema incluyendo circuitos integrados, terminales de video, impresora, mandos de juego, y dispositivos auxiliares de memoria.

herencia: Una relación entre clases en donde una clase comparte comparte la estructura y la conducta definida en una (herencia simple) o varias (herencia múltiple) clases. La herencia define jerarquía del tipo "una clase de" entre clases en la cual una subclase hereda de una o más superclases; una subclase típicamente aumenta o redefine la estrcutura y la conducta existente en su superclase.

identidad: La naturaleza de un objeto que lo distingue de todos los demás objetos.

identificador: El nombre, dirección, etiqueta o índice distintivo de un objeto en un programa.

implantación: En la POO, vista interior de un objeto, clase o módulo, incluyendo las características elementales (secretas) de su conducta.

indecible: (problema) Dícese de aquellos problemas que no poseen una solución algorítmica

independencia: Término usado para describir pares de módulos: Se dice que dos módulos son independientes si cada unp puede funcionar completamente sin la presencia del otro.

instancias: Un ejemplo de clase. Una instancia posee estado, conducta e identidad. La estructura y la conducta de instancias similares están definidas en su clase común. Los términos de instancia y objeto son intercambiables

insumos: Los datos de los cuales se alimenta un sistema

integración de sistemas: Fase del desarrollo tradicional de sistemas. Cuando ya se poseen codificados todos los módulos del sistema, o todos los subsistemas (los cuales han sido desarrollados en forma paralela por distintos programadores) se procede a unirlos en un solo sistema. A esta etapa de unión se le conoce como integración de sistemas y culmina cuando todos los subsistemas interactúan correctamente.

jerarquía: Un rango de ordenemiento de la abstracción. Las dos más comunes jeraquías de un sistema complejo incluyen su estructura de clase (la jerarquía "una clase de") y su estrcutura de objeto (La jerarquía "es parte de"); las jerarquías también son encontradas en los módulos y procesos arquitectónicos de un sistema complejo. La jerarquía es uno de los elementos del modelo de objetos.

lista de requierimientos: Es la lista de las necesidades y funciones que deberá cubrir un sistema que se encuentre en su fase de desarrollo. La lista suele cubrir también las especificaciones sobre el tiempo de desarrollo y el equipo de se utilizará.

macro: Un módulo cuyo cuerpo es copiado efectivamente en linea durante la traduccion (sea compilación o ensamble) como resultado de la invocación de un nombre; esto es, el contenido reemplaza a la referencia del identificador agregado.

mantenimiento: La corrección de errores que son descubiertos en el sistema durante su vida productiva

máquina de Turing: Modelo matemático (no es una máquina) de un autómata general. Alan M. Turing lo diseñó como parte de sus avanzados estudios sobre computabbilidad y algoritmos, y se emplea como medio de análisis en el trabajo teórico, y como herramientas para probar teoremas y proposiciones en matemáticas computacionales y teoría de lenguajes.

mensaje: Una operación que un objeto realiza a través de otro. Los términos mensaje de método y operación son intercambiables.

método: Vease: función miembro

modelo: Aquello que sirve para representar o describir otra cosa; un prototipo. El modelo puede tener una forma semejante (un automóvil de madera) o ser totalmente distinta del objeto real (un modelo matemático). Los modelos se utilizan para estudiar y anticipar el comportamiento de un dispositivo más grande y funcional.

modelo de objeto: La colección de los principios que fundamentan el diseño orientado a objetos; un paradigma de software que enfatiza los principios de abstracción, encapsulamiento, jerarquía, tipificación, concurrecia y persistencia.

modelo matemático: Representación matemática de un proceso, dispositivo o concepto que permite la manipulación matemática de variables a fin de estudiar el comportamiento del proceso, dispositivo o concepto en diferentes condiciones.

modularidad: La propiedad de un sistema que tiene de ser descompuesto dentro de un conjunto modulos cohesivos y débilmente acoplados. La modularidad es uno de los elementos funcionales del modelo de objeto.

multiproceso: Se dice que existe multiproceso cuando se tienen varios programas funcionando simultáneamente.

multiusuario (sistema): Sistema que permite a más de un usuario tener acceso a todos los recursos del sistema, al mismo tiempo.

modulo: En su forma más genérica, una secuencia contigua de sentencias de programa limitadas por elementos limitadores, que poseen un identificador. Un módulo posee típicamente una interface y una implantación

nivel de abstracción: El rango de abstracción en una estructura de clases, estrcutura de objeto, arquitectura de módulos, o proceso arquitectónico. En términos de jerarquía como "parte de", una abstracción dada es un nivel de alta abstracción cuando unos se construyen a partir de otras; en términos de jerarquía como "una clase de", un alto nivel de abstracción son las generalizaciones, y un bajo nivel se encuentra en la especialización.

nodo: Unidad de información compuesta por elementos básicos. Un punto de información dentro de una estructura de mayor complejidad.

objeto: Un ejemplo de clase. Una instancia posee estado, conducta e identidad. La estructura y la conducta de instancias similares están definidas en su clase común. Los términos de instancia y objeto son intercambiables

ocultamiento de información: Diseño heristico desarrollado por D.L. Parnas: Los módulos son formados de tal manera que ocultan su información al resto del sistema.

orientado a los datos: Dicese de aquellos sistemas o metodologías cuyo principal punto de referencia son los datos de un sistema. Su premisa básica asume que un programa debe ser escrito alrededor de los datos ya que los resultados giran alrededor de ellos.

orientado a los procedimientos: Dicese de aquellos sistemas o metodologías cuyo punto de referencia es el programa. Su premisa básica asume que los programas y los procedimientos manejan a los datos y por lo tanto, si se quiere un correcto manejo de los datos, debe manejarse en forma correcta a los elementos que controlan a los datos (es decir, los programas).

parametro: Lista de variables que envía durante una llamada un programa a una rutina. Variable a la que se le asigna un valor constante para un fin específico. También, medida de valor que delimita un proceso.

polimorfismo: Un concepto en la teoría de tipos, acorde al nombre (tal como una declaración de variable) puede denotar objetos de muchas clases diferentes que están relacionadas por una superclase común; cualquier objeto denotado por este nombre, es capaz de responder a un conjunto de operaciones comúnes de diferente forma.

portabilidad: Una propiedad de los programas que representa su fácil movimiento entre distintos ambientes de trabajo.

privado: Una declaración que forma parte de la interface de una clase, objeto o módulo; que es declarad como privada no es visible a cualquier otra clase, objeto o módulo.

problema indecible: Vease indeble problema

procedimental: vease orientado a los procedimientos

programa: Conjunto detallado y explicito de instrucciones de computadora para realizar algún trabajo. Un programa se escribe en un lenguaje adecuado para la entrada de la computadora; es completo e independiente.

programa propio: Un programa propio es aquel que posee un solo punto de entrada por arriba, se lee de arriba hacia abajo y tiene un solo punto de salida por abajo.

programación: Ciencia de planear la solución de llos problemas al reducir el plan a un conjunto de instrucciones razonables para la máquina que dirige las acciones de un sistema de computación o procesamiento de datos. La computadora realiza operaciones aritmétiocas y lógicas que dan solución a los problemas.

programación estructurada: Un conjunto de principios y técnicas para la escritura de programas como un conjunto anidado de entradas y salidas únicas de bloques de código; usando un restringido número de estructuras de control

programación orientada al objeto: Un método de programación en el cual los programas son organizados como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de algún tipo, y cuyos tipos son todos miembros de una jerarquía de tipos unidos vía otras relaciones heredables. En tales programas, los tipos son generalmente vistos como estáticos, en tanto que los objetos poseen una naturaleza mucho más dinámica, la cual se encuentra dada por la existencia del enlace dinñamico y el polimorfismo.

programador: Un término informal usado para describir a la persona que diseña y escribe la instrucciones de programación para implantar un módulo. En algunas organizaciones, los programadores son responsables del diseño estrcutural del sistema, y ocasionalmente también del análisis de requerimientos.

protegido: Una declaración que forma parte de la interface de la interface de una clase, objeto o módulo, que no es visible a otra clases, objetos o módulos, excepto a aquellos que representen subclases.

pruebas: Un proceso de demostración de que el sistema funciona en base a las especificaciones

pseudocódigo: Conjunto de ordenes escritas en español. Cada orden consiste de un verbo un un objeto sobre al cual actúa el verbo.

público: Una declaración que forma parte de la interface de una clase, objeto o módulo, y que es visibble para todas las otras clases, objetos o módulos

puntero: Una entidad conteniendo o teneiendo una referencia de un identificador.

punto fijo: Punto en un número en que todos los numerales de la izquierda son enteros y todos los de la derecha son fraccionarios. El programador no ubica el punto aritmético, sino que permanece fijo en una posición determinada.

punto flotante: En notación de punto flotante, un número se expresa como un numeral de punto fijo que sirve como coeficiente de multiplicación y una parte exponencial formada por la raíz del sistema de numeración elevada a una potencia que ubique el punto aritmético; de ahí el término punto flotante. Por ejemplo, un número de punto flotante en el sistema de base 10 será 1.34 x 102 igual a 134.

refinamiento sucesivo: Es una metodología en la cual se realiza una especificación de requerimientos inicial y superficial. En base a esta lista se diseña una primera versión del sistema. Tan pronto como se tiene este primer diseño, se realiza una lista más detallada de requerimientos y se procede a refinar el diseño previo para que cumpla con las nuevas especificaciones. Este proceso se repite tantas veces como sea necesario.

sentencia: Una línea, instrucción u otra construcción bien definida de un lenguaje de programación que describe o dirige un paso o una parte de la solución de un problema.

servidor: Un objeto que nunca opera sobre otros objetos, pero que que es solo operado por otros objetos.

sistema de información: En una organización el conjunto total de procedimeintos, operaciones y funciones dedicados a la generación, recolección, evaluación, almacenamiento, recuperación y difusión de datos e información.

sobrecarga: Asignación de más conductas a un operador o a una función además de las que esta puede manejar por omisión.

software: Programs y rutinas (instrucciones secuenciales) que indican a la computadora que hacer y cuando hacerlo. También denota la documentación: manuales, diagramas e instrucciones del operador.

subclase: Es una clase que depende de otra para quedar totalmente definida

superclase: Es una clase que sirve de base para que otras clases puedan ser definidas

tabla de decisión: Una tabla de decisión es una forma compacta que muestra la o las acciones que deben seguirse cuando haya varias combinaciones de condiciones.

tiempo de compilacion: Es aquel tiempo en el cual el programa fuente pasa por el proceso de compilación.

tiempo de corrida: Ver tiempo de ejecucion

tiempo de ejecucion: Es aquel tiempo en el cual un programa se encuentra ejecutando las funciones para las cuales fué programado.

tipo: La definición del dominio de los valoresposibles que un objeto puede poseer y el conjunto de operaciones válidas que pueden operar sobre ese objeto.

top-down: Vease Diseño descendente

usuario: Operador final de una computadora.

APÉNDICE D

Referencias

Barkakati, Naba 1991. *Object-Oriented in C++*. 1a reimpresion. Carmel, Indiana.:SAMS. 666p.

Bobrow, D, y Stefik, M. Febrero 1986. Perspectives on Artificial Intelligence Programming. *Science*, vol 231, p951

Booch Grady. 1991. *Object Oriented Design with Applications*. The benjamin/Cummings Publishing Company. 580p

Budd Timothy. 1991. *An Introduction to Object-Oriented Programming*. Reading Massachusetts: Addison-Wesley. 399p

Corrado Bohm y Giuseppe Jacopini.1966. Flow diagramsm, Turing Machines and Leanguages with Only Two Formulatión Rules. *Communications of ACM, vol 9, No 5* (Mayo 1966) pp. 366-371.

Cristie Linda y John. 1986. *Enciclopedia de términos de microcomputación*. 1 reimpresión. México D.F. Prentice Hall. s.p

Dahl,O, Dijkstra,E y Hoare,C.A.R. 1972. *Structured Programmign*. Londres,Inglaterra: Academic Press

Eckel Bruce. 1991. *Aplique C++*.Madrid España. Mc Graw Hill. 521 p

Fairley Richard. 1990. *Ingeniería de sotfware*. México D.F.: Mc Graw Hill. 390p

Kernighan B, y Ritchie Dennis. 1985. *El lenguaje de programación C*. 5a reimpresion. México .D.F. Prentice Hall. 235p

Levine Guillermo.1992. *Introducción a la computación y a la programación estructurada*. Mexico D.F. : Mc Graw Hill. 424p

Miller,G, Marzo 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The psychological Review*, vol 63, P. 86

Pressman, Roger S. 1988. Ingenieria del software. Un enfoque práctico. 2a.Edición D.F. México:Mc Graw Hill. 628p

Rentsch, T. Septiembre 1982 Objetc-Oriented Programming. SIGPLAN Notices. vol 17

Senn, James A. 1991. *Análisis y diseño de sistemas de información*. 2a Edición. Mexico D.F. :Mc Graw Hill. 942p.

Schild Herbert. 1990. *Manual de referencia C*. Madrid, España. Mc Graw Hill. 749p.

Wirfs-Brock R., Wilkerson B. y Wiener L. 1990. *Designing Objet-Oriented Software.* Englewood, NJ.:Prentice Hall. 341p

Wirth,N. 1986. *Algorithms and Data structurs* . Englkewood Cliffs,NJ:Prentice Hall.

Yourdon,E.,y Constantine,L. 1979. *Structured Design.* Englewood Cliffs,NJ:Prentice Hall. 473p.

Yourdon,E. 1975. *Techniques of Program Structure and Design.* Englewood Cliffs,NJ:Prentice hall