Inicialízate en la Programación

con C++

La habilidad para programar ayuda a muchos profesionistas de la actualidad a ser más competitivos, ya que les permite sacar mayor provecho de las capacidades de las computadoras y ponerlas a su servicio, sin embargo, no todos los libros de programación son suficientemente "digeribles" para quienes no se han iniciado en este arte. Este libro contiene ejercicios de programación estructurada, cuya solución se va presentando poco a poco al lector para incentivarle a proporcionar sus propias soluciones y compararlas con las que se le brindan. Además de la descripción del tema, se incluyen varios ejemplos resueltos sobre el mismo, para que el lector aprenda a través de la observación de los ejemplos y de la resolución de problemas similares a problemas previos.

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda
Con la colaboración de
Pedro Pablo González Pérez

UNIVERSIDAD AUTÓNOMA METROPOLITANA
CUAJIMALPA

9 786072 800786

Inicialízate en la Programación

UNIVERSIDAD AUTÓNOMA METROPOLITANA

CUAJIMALPA

Inicialízate en la Programación

con C++

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda
Con la colaboración de
Pedro Pablo González Pérez

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA
UNIDAD CUAJIMALPA

DIVISIÓN DE CIENCIAS NATURALES E INGENIERÍA

INICIALÍZATE EN LA PROGRAMACIÓN CON C++

AUTORES:
Dra. María del Carmen Gómez Fuentes
Dr. Jorge Cervantes Ojeda
Dr. Pedro Pablo González Pérez (colaborador)

Departamento de Matemáticas Apoicadas y Sistemas

ISBN: 978-607-28-0078-6

Diciembre 2013

UNIVERSIDAD AUTÓNOMA METROPOLITANA

Inicialízate en la Programación

con C++

Autores:

Dra. María del Carmen Gómez Fuentes

Dr. Jorge Cervantes Ojeda

Colaborador:

Dr. Pedro Pablo González Pérez

Editores
María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Departamento de Matemáticas Aplicadas y Sistemas.
División de Ciencias Naturales e Ingeniería
Universidad Autónoma Metropolitana, Unidad Cuajimalpa

Editada por:
UNIVERSIDAD AUTONOMA METROPOLITANA
Prolongación Canal de Miramontes 3855,
Quinto Piso, Col. Ex Hacienda de San Juan de Dios,
Del. Tlalpan, C.P. 14787, México D.F.

Inicialízate en la Programación con C++

Primera edición 2013

ISBN: 978-607-28-0078-6

Impreso en México
Impreso por Publidisa Mexicana S. A. de C.V.
Calz. Chabacano No. 69, Planta Alta
Col. Asturias C.P.

Contenido

Prefacio

La transmisión de un pensamiento tan abstracto como lo es la programación requiere de la propuesta de nuevas estrategias en las que se involucren de forma equilibrada la teoría, los ejemplos y la práctica. En este libro presentamos un método de enseñanza de la programación basado en el constructivismo cuyo objetivo principal es ayudar a aquellos que se enfrentan a la programación por primera vez.

La habilidad para programar es una necesidad actual de los profesionales que les permite ser competitivos. Y no hablamos solamente de los profesionales de la informática, sino también de otros especialistas que requieren de la programación para poner en práctica sus conocimientos y hacerlos de utilidad tales como ingenieros, matemáticos, físicos, entre muchos otros. Hemos detectado a lo largo de doce años de experiencia impartiendo cursos de programación estructurada, que a muchos de los alumnos que se enfrentan por primera vez al reto de programar una computadora se les dificulta entender en qué consiste, puesto que requiere de un conocimiento abstracto que no tiene similitudes con el de otras disciplinas. Algunas veces tienden a memorizar y a mecanizar los problemas, lo que les impide adquirir la habilidad de solucionar problemas de diversa índole.

Los programas de ejemplo son una rica fuente de información para presentar, analizar y discutir un lenguaje de programación, sin embargo, transmitir las estrategias para crear estos programas es mucho más complicado. Hemos notado la necesidad de que los alumnos cuenten con un libro que contenga, además de la descripción del tema, varios ejemplos resueltos sobre el mismo, con el fin de aprender de la observación de los ejemplos, y de la resolución de problemas similares a problemas previos. Procuramos que los ejercicios incluidos apoyen el aprendizaje de manera que sean un reto, pero situados en una zona asequible de acuerdo con el nivel de conocimientos y habilidades adquiridos. Con fundamento en la *zona de desarrollo próximo* planteada por Vygotsky, propusimos los ejercicios de tal forma que se conviertan en un andamiaje para el alumno de acuerdo con su nivel de avance. Este andamiaje le permite sostenerse y se puede ir quitando una vez que se han adquirido las habilidades y conocimientos necesarios para la resolución autónoma de los problemas.

La metodología de este libro tiene el objetivo de guiar al estudiante en su aprendizaje de la programación estructurada y le llamamos *Solución por Etapas* porque consiste en mostrar una o varias etapas de la solución de cada ejercicio hasta llegar a la solución completa. Queremos lograr que el alumno resuelva los problemas sin necesidad de ver la solución, sin embargo, en caso de que necesite una ayuda para continuar o para empezar a resolverlo, puede consultar la siguiente etapa sin necesidad de ver la solución completa. Pretendemos motivar a los estudiantes a que resuelvan los ejercicios dándoles la confianza de que no están solos, ya que saben que en el momento en el que no puedan continuar, pueden consultar la ayuda que se les brinda por etapas. De esta forma todos los *ejercicios* son realmente ejemplos y es el alumno el que decide cuanto estudia la solución planteada y cuanto se esfuerza en solucionar el problema por su cuenta. Esto permite al estudiante

tener una sensación de control de la tarea y aumentar su eficacia lo cual es un requisito fundamental para ejercer un esfuerzo constante en el aprendizaje. La comparación de los ejercicios resueltos con la solución propia del alumno, brinda tres beneficios importantes: i) el alumno adquiere el conocimiento de cómo se escribe un programa con el formato adecuado, es decir, el uso correcto de sangrías, espacios, comentarios, nombres de variables, etc., ii) por medio de la observación consciente, promovida de forma explícita, adquiere experiencia en cuanto a la optimización de los algoritmos, y iii) demostración en la práctica de que existen diferentes alternativas para resolver un problema. Si el alumno resolvió correctamente el ejercicio, profundizará su conocimiento al observar y entender otra manera de resolverlo. Por otra parte, al observar las soluciones el alumno practica la lectura de código con lo cual adquiere familiaridad con los programas y con la forma en la que posiblemente lo escribiría un programador con experiencia.

Dedicamos el capítulo II *Estructuras de control* a la iniciación en la programación. En este capítulo pretendemos que el alumno se sienta capaz de resolver el siguiente problema, proporcionando varios problemas del mismo tipo, aunque al principio sólo sea por imitación. Resolver varios problemas de un mismo tipo le brinda seguridad y confianza en el tema. Además, adquiere un mayor nivel de abstracción al ver cómo un mismo concepto puede aplicarse en diferentes contextos. Se pretende estimular al estudiante a que *aprenda haciendo* y que tenga disponibles ejemplos suficientes para aprender los temas que le ofrezcan dificultad. Esto permite que los materiales se ajusten al marco de referencia que posee el alumno de manera que se adueñe de ellos y use la información de modo compatible con lo que ya sabe. Con los ejercicios resueltos por etapas pretendemos que el alumno practique y se autoevalúe de dos formas opcionales: la primera es que vaya comparando su solución parcial con cada una de las etapas que van solucionando el problema, esto se recomienda al principio, cuando el tema es nuevo; y la segunda manera es resolver el problema por completo y autoevaluarse con la solución final, esto es recomendable cuando ya se ha comprendido mejor el tema o cuando el alumno decide asumir un reto mayor. En el resto de los capítulos se presentan con ejemplos y se dedica una sección a los ejercicios,

Es importante mencionar que lo que se pretende con los ejercicios en los que se aplica la metodología de *Solución por Etapas*, no es una "mecanización de la programación", sino más bien la comprensión de los ejemplos y la abstracción de las ideas importantes. Por ello consideramos que unos cuantos problemas hechos de manera razonada darán una buena base para la programación. Los ejercicios incluidos en la literatura actual pueden servir para reafirmar los conocimientos adquiridos. Los programadores novatos no sólo deben aprender a resolver los problemas y convertir la solución en un programa, sino que deben tener habilidad para dar seguimiento y depurar sus programas. Por lo tanto es indispensable que la enseñanza se complemente con el laboratorio.

¿Por qué elegimos C++ para la enseñanza de la programación estructurada?

Hay lenguajes de programación muy populares para este propósito, tales como Pascal, Fortran, Basic, C, etc., y hay buenas razones para elegir a cada uno de ellos, pero también hay buenas razones para elegir C++. Optamos por C++ porque es un lenguaje poderoso, que, en caso de que el alumno decida avanzar a niveles más altos de la programación, tiene todo para desarrollar programas con las técnicas más avanzadas. También hemos podido observar entre nuestros alumnos, que existe una cierta predilección por el primer lenguaje de programación aprendido. Si el alumno tiene C++ como primer lenguaje de programación, no le será nada difícil aprender Java o C, los cuales tienen una sintaxis muy parecida y son lenguajes muy usados en la industria del software. Además le será más fácil aprender lenguajes de alto nivel como Pascal o Basic.

La capacidad reducida para retener y procesar información a corto plazo, es uno de los principales límites de la mente, por lo que es importante optimizar el uso de la memoria a corto plazo durante el proceso de enseñanza-aprendizaje. La información nueva que se le presenta a los alumnos debe ser concisa, breve y lo más objetiva posible. Un curso sobre cualquier materia, debe

concentrarse en unas pocas ideas centrales o generales y adentrarse en otras de menor importancia según se disponga de tiempo. A nivel básico, trabajar con algunas de las instrucciones del lenguaje C++, es mucho más sencillo que trabajar con las instrucciones equivalentes del lenguaje C, por ejemplo, las instrucciones de entrada y salida de datos. Las operaciones de entrada/salida en lenguaje C requieren de más tiempo y memoria a corto plazo de lo que se requiere con C++. A nuestro juicio, no vale la pena invertir mucho esfuerzo en el aprendizaje de las operaciones básicas de entrada/salida de datos, ya que, en la práctica, una vez que alguien domina la programación, trabaja con ambientes de desarrollo en los que el programador puede generar rápidamente interfaces de usuario gráficas avanzadas sin necesidad de recurrir a las instrucciones básicas. Es decir, las instrucciones para operaciones de entrada y salida de datos del usuario que se exponen en los libros para principiantes, tienen un fin únicamente didáctico y, por lo tanto, temporal. Así que debemos usar el menor tiempo posible en este tema. Otro ejemplo es, el manejo de archivos con C++ que también es muy simple. Con C++ es posible abordar el tema desde los primeros capítulos y después hacer ejercicios de entrada/salida tanto en pantalla como en archivos a lo largo de todo el curso. Creemos que es importante que los estudiantes se familiaricen bien con el acceso a la información en disco, pues esto les será muy útil en el futuro. En general, C++ nos permite avanzar rápidamente en el tema de entrada/salida de datos que, a nivel básico no requiere de entrar en muchos detalles, mientras que en C se le exige al estudiante un esfuerzo mayor y, a nuestro juicio, inútil.

Esperamos que este libro sea de utilidad a todos aquellos que deseen aprender a programar.

María del Carmen Gómez.

Jorge Cervantes

Agradecimiento

Agradecemos profundamente la valiosa colaboración del Dr. Pedro Pablo González Pérez en el capítulo I "Introducción" y en el capítulo VII "Búsqueda y ordenamiento" la cual enriqueció considerablemente el contenido de estos capítulos.

María del Carmen Gómez.

Jorge Cervantes.

Capítulo I Introducción

Pedro Pablo González Pérez
María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

- Proporcionar una visión panorámica de las estructuras y de los conceptos básicos de la programación estructurada
- Introducir al lector al lenguaje de programación C++

I.1 Concepto de Algoritmo y sus propiedades

Un algoritmo es un procedimiento para resolver un problema dado. Un algoritmo se describe con un conjunto finito y ordenado de pasos que deben llevarse a cabo para producir la solución a dicho problema (ver Figura I-1).

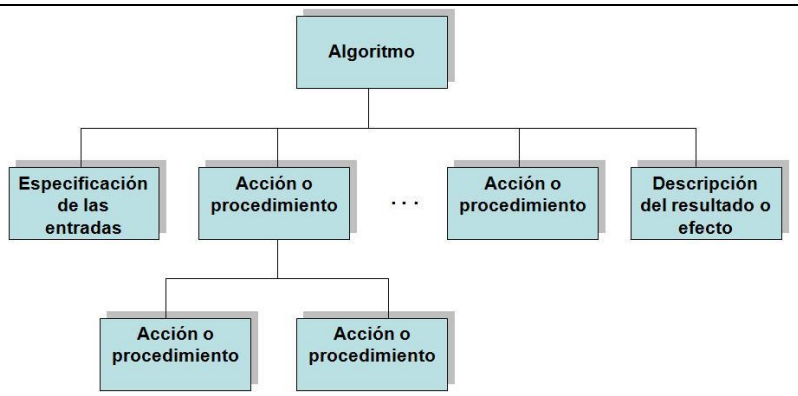


Figura I-1 *Representación gráfica de la estructura de un algoritmo.*

La completa ejecución de un algoritmo debe finalizar con la producción del resultado esperado a partir de las entradas proporcionadas (ver Figura I-2).

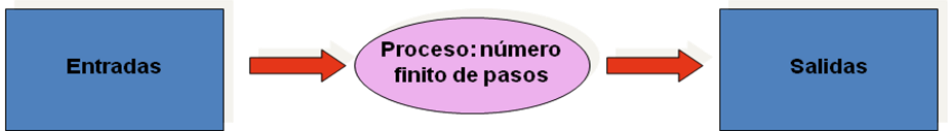


Figura I-2 *Los tres componentes clave de un algoritmo: entradas, número finito de pasos y salidas.*

La palabra "*algoritmo*" (que a veces se confunde con la palabra "logaritmo"), es una palabra relativamente nueva, apareció en el diccionario de Webster de palabras nuevas hasta 1957. Hasta antes de esta fecha se conocía la palabra "*algorismo*" que significa el proceso de hacer aritmética utilizando los números arábigos. En la edad media, los "*abacistas*" utilizaban el ábaco para hacer sus cálculos, mientras que los "*algoristas*" calculaban por medio del "*algorismo"*.

El significado moderno de la palabra "*algoritmo*" es muy similar al de: método, proceso, técnica, procedimiento, rutina. Sin embargo, la palabra "*algoritmo*" además de representar únicamente un conjunto de operaciones en secuencia para resolver un problema específico, se caracteriza por tener las propiedades que se describen en seguida.

Los algoritmos tienen las siguientes propiedades:

1. ***Entradas y salidas***. Se debe definir en forma clara cuáles son las entradas que se requieren. También deben especificarse los resultados que el algoritmo debe producir.
2. ***Precisión***. Se debe indicar sin ambigüedades el orden exacto de los pasos o acciones a seguir en el algoritmo y la manera en que estos pasos deben realizarse.
3. ***Número finito de pasos***. El número total de pasos que se ejecutan, por cada conjunto de datos de ingreso, debe ser finito, es decir, el algoritmo debe poder terminar en algún momento.
4. ***Determinismo***. Para un mismo conjunto de datos proporcionado como entrada, el algoritmo debe producir siempre el mismo resultado.
5. ***Efectividad***. Cada paso o acción se debe poder llevar a cabo en un tiempo finito.

Se dice que dos algoritmos son equivalentes cuando se cumplen todas las condiciones siguientes:

- Poseen el mismo dominio de entrada.
- Poseen el mismo dominio de salida.
- Para los mismos valores del dominio de entrada producen los mismos valores en el dominio de salida.

A continuación se presentan dos ejemplos de algoritmo (Ejemplo 1.1 y Ejemplo 1.2). El primer algoritmo resuleve el problema del cálculo del área y el perímetro de un rectángulo. El segundo algoritmo calcula la edad y peso promedio de un grupo de estudiantes. Estos ejemplos pretenden ilustrar el uso del algoritmo en la solución de problemas sencillos e iniciar la familiarización del lector con la formulación de los mismos. Las estructuras de asignación, selección y control comúnmente utilizadas en un algoritmo, se exponen a lo largo de este capítulo y se estudian con más detenimiento en el capítulo II.

Ejemplo 1.1.- Calcular el área y el perímetro de un rectángulo.

Datos de entrada:

`base` y `altura`

Datos de salida:

`área` y `perímetro`

Operaciones a ejecutar:

`area` resulta de `base*altura`
`perímetro` resulta de `2*(base + altura)`

Algoritmo:

```
Inicio
  Leer (base, altura)
  area          vale base*altura
  perimetro     vale 2*(base + altura)
  Escribir "Área del rectángulo es: ", area
  Escribir "Perímetro del rectángulo es: ", perimetro
Fin
```

Ejemplo 1.2.- calcular la edad y peso promedio de un grupo de N estudiantes

Datos de entrada:

`NumTotal` (número total de estudiantes)
`Edad` (para cada estudiante)
`Peso` (para cada estudiante)

Datos de salida:

`EdadPromedio`
`PesoPromedio`

Operaciones a ejecutar:

Pedir el número total de estudiantes: `NumTotal`
Pedir todas las edades de los estudiantes y calcular la suma: `SumaEdades`
Pedir todos los pesos de los estudiantes y calcular la suma: `SumaPeso`
`EdadPromedio` resulta de `SumaEdades/NumTotal`
`PesoPromedio` resulta de `SumaPeso/NumTotal`

Algoritmo:

```
Inicio
  Leer (NumTotal)

  SumaEdades  vale 0
  SumaPeso     vale 0

  Para cada valor de contador desde 1 hasta NumTotal
    Leer (Edad)
    SumaEdades vale SumaEdades + Edad
    Leer (Peso)
    SumaPeso  vale SumaPeso + Peso
  finPara

  EdadPromedio  vale SumaEdad/NumTotal
  PesoPromedio  vale Sumapeso/NumTotal

  Escribir "La edad promedio es:", EdadPromedio
  Escribir "El peso promedio es:", PesoPromedio
fin
```

I.2 Especificación de algoritmos en computadora

La resolución de un problema en computadora requiere como paso previo el diseño de un algoritmo que especifique el procedimiento para resolver el problema. Es decir, antes de codificar un programa se requiere diseñar el algoritmo, a su vez, el diseño del algoritmo requiere de un análisis y de la descripción del problema (ver Figura I-3).

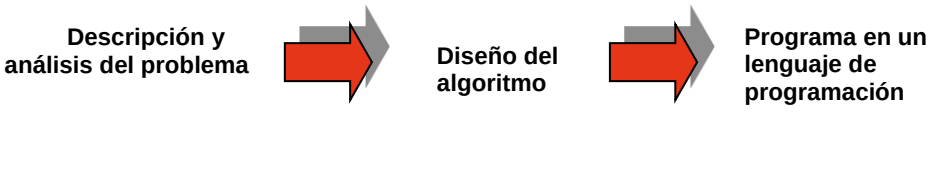


Figura I-3. El diseño del algoritmo como paso previo a la codificación del programa

I.2.1 Especificación de un algoritmo a través del pseudocódigo

Una de las formas de especificar un algoritmo es mediante la escritura en lenguaje natural de una secuencia de pasos numerados. El regreso a un paso anterior o el salto a un paso posterior se indica con palabras. Otra de las formas de especificarlo es mediante el *pseudocódigo*, en éste se utiliza un lenguaje estructurado y cercano a lo que será finalmente el programa de computadora (en los ejemplos I.1, I.2 se utilizó pseudocódigo para la especificación de los algoritmos).

El pseudocódigo es un lenguaje para la especificación de algoritmos con una sintaxis para las estructuras de control similar a la expresada en un lenguaje de programación. Traducir un algoritmo escrito en pseudocódigo a un lenguaje de programación resulta más sencillo que hacer la traducción desde un algoritmo expresado en lenguaje natural.

Las acciones y las estructuras de control se representan en el pseudocódigo con palabras reservadas, similares a las utilizadas en los lenguajes de programación estructurada. Entre estas palabras reservadas, las más usadas en inglés son: **begin**, **read**, **write**, **if-then**, **if-then-else**, **while-end**, **do-while**, **repeat for-to** y **end**. En español se utilizan palabras como las siguientes: **inicio**, **leer**, **escribir**, **si-entonces**, **si-entonces-sino**, **mientras-hacer**, **hacer-mientras**, **repetir desde-hasta** y **fin**. Es muy recomendable utilizar solo una de estas dos formas durante la escritura de un algoritmo, ya que la mezcla o uso indistinto de palabras procedentes del inglés y palabras procedentes del español puede causar confusión y dificultad en la lectura e interpretación del algoritmo.

Cuando se usa el pseudocódigo como lenguaje de especificación de un algoritmo, el programador puede concentrarse en la lógica y en las estructuras de control, sin preocuparse por la sintaxis y reglas del lenguaje de programación.

Ejemplo 1.3.- . A continuación presentamos, en pseudocódigo, otra versión del algoritmo para el cálculo de la edad y peso promedio de un grupo de estudiantes previamente visto en el Ejemplo 1.2.

```
inicio
   Leer (NumTotal)
   i ← 0
   SumaEdades ← 0
   SumaPesos ← 0
   mientras i < NumTotal
      leer (Edad, Peso)
      SumaEdades ← SumaEdades + Edad
      SumaPesos ←SumaPesos + Peso
      i ← i + 1
   finMientras

   EdadPromedio ← SumaEdades/NumTotal
   PesoPromedio ← SumaPesos/NumTotal
   Escribir  "La edad promedio es: ", EdadPromedio
   Escribir  "El peso promedio es: ", PesoPromedio
fin
```

Nótese que aquí se usaron flechas para indicar que el resultado de una operación es usado como el valor asignado a una *variable*.

I.2.2 Diagrama de flujo: representación gráfica del algoritmo.

Un diagrama de flujo es una técnica de representación gráfica de los pasos de un algoritmo. El diagrama de flujo consiste de un conjunto de símbolos, tales como rectángulos, paralelogramos, rombos, etc. y flechas que conectan estos símbolos. Los símbolos representan las diferentes acciones que se pueden ejecutar en un algoritmo (lectura, asignación, decisión, escritura, etc.), mientras que las flechas muestran la progresión paso a paso a través del algoritmo.

Los diagramas de flujo deben mostrar siempre los símbolos correspondientes al "inicio" y al "fin" del algoritmo. La secuencia entre los pasos se indica por medio de flechas, mientras que existen diferentes símbolos para representar la ejecución de ciclos y la toma de decisiones. Si se hace un diagrama de flujo correctamente, la traducción del algoritmo a un lenguaje de programación resulta relativamente sencilla. El diagrama de flujo es de gran ayuda para visualizar la lógica del algortimo. En Tabla I-1 se incluyen los símbolos que se utilizan en un diagrama de flujo.

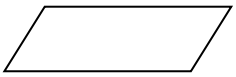
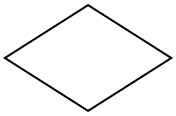
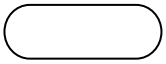
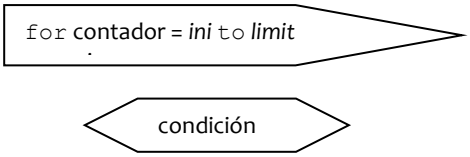
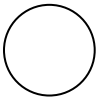
Símbolo	Representación gráfica	Significado
Flechas o líneas de flujo		Indica el sentido de ejecución de las acciones
Rectángulo		Acción a realizar. Operación que se lleva a cabo.
Paralelogramo		Entrada de información
impresión		Salida de información
Rombo		Selección o decisión lógica
Terminal		Inicio y fin del diagrama
Símbolos de iteración		Para indicar ciclos
Círculo		Se usa como conector entre dos partes del diagrama

Tabla I-1 Símbolos comúnmente utilizados en la construcción de un diagrama de flujo.

En la Figura I-4 se muestra un ejemplo de un diagrama de flujo que representa gráficamente al algoritmo para el cálculo de la edad y peso promedio de un grupo de estudiantes, cuyo pseudocódigo vimos en el ejemplo 1.3. En la sección 0 se mostrará este mismo algoritmo pero usando una estructura iterativa con control posterior.

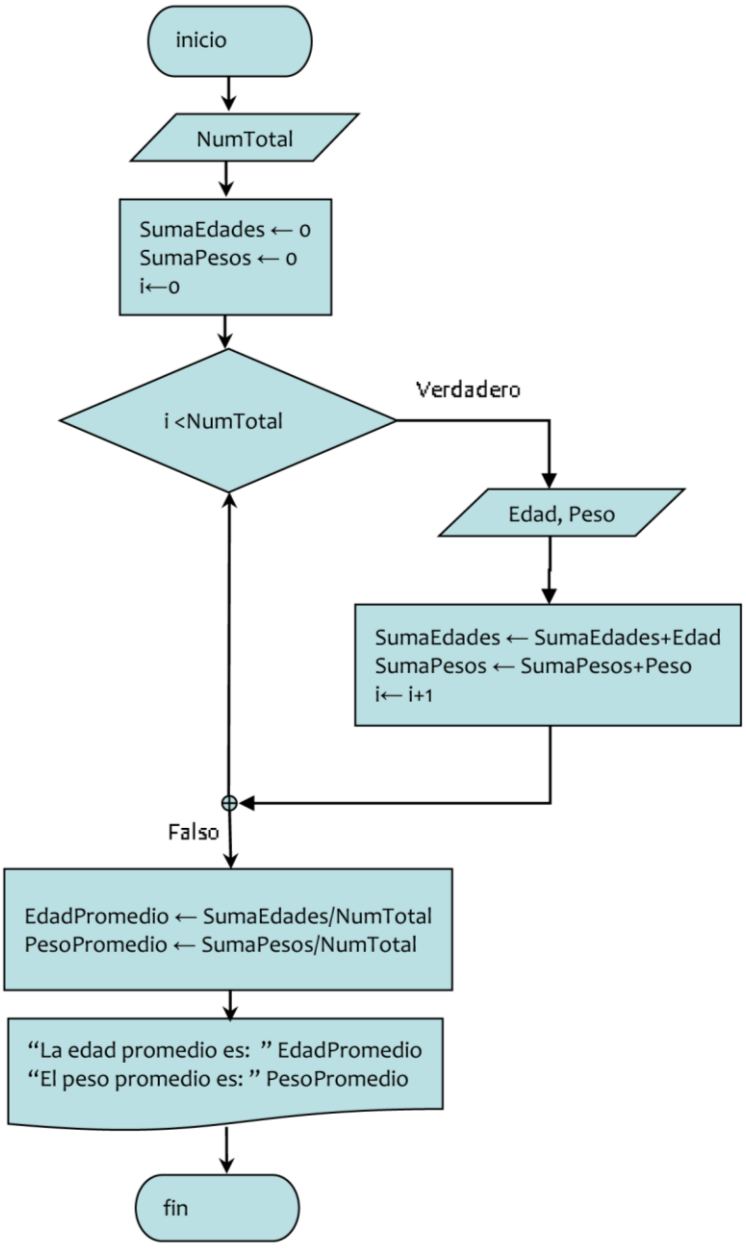
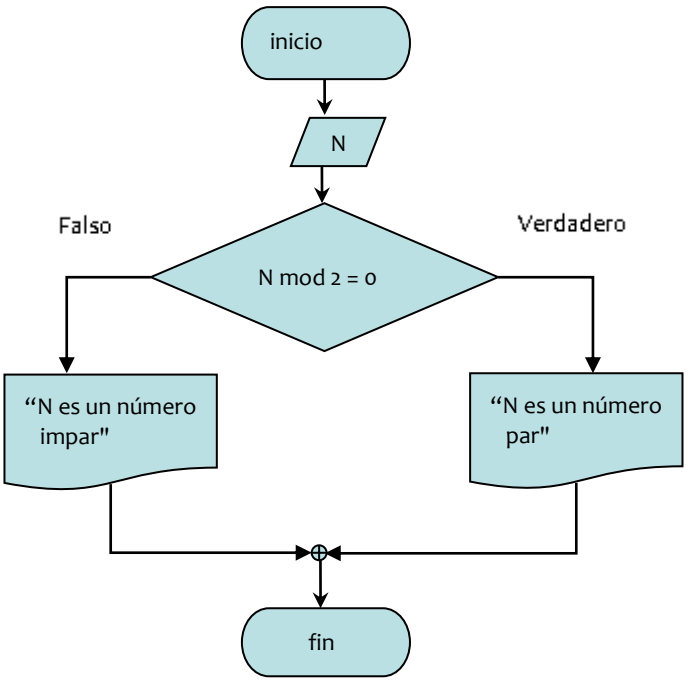


Figura I-4 ***Diagrama de flujo correspondiente al algoritmo para el cálculo de la edad y peso promedio de un grupo de alumnos.***

En ocasiones es posible proponer varios algoritmos para resolver un mismo problema, es decir, disponer de dos o más algoritmos que sean equivalentes. Cuando este es el caso, es necesario comparar las alternativas propuestas y elegir la más adecuada. Por ejemplo en la Figura I-5 se proponen dos algoritmos (en su representación gráfica) para determinar si un número es par o impar.



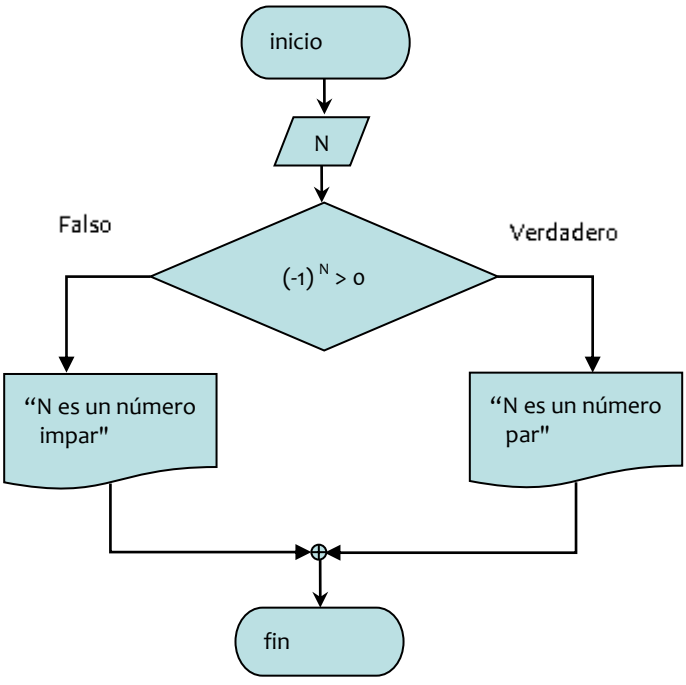


Figura I-5 *Dos algoritmos diferentes para determinar si un número N es par o impar.*

Otros ejemplos en los que es posible proponer algoritmos equivalentes para el problema a resolver son los siguientes:

- Obtener la raíz cuadrada de un número.

- Cálculo del mínimo común denominador entre dos números enteros M y N.

Cuando existen varias formas de resolver un mismo problema, será necesario determinar cuál resulta ser la más eficiente. La disciplina denominada "*Análisis de algoritmos*" se encarga de estudiar la eficiencia de los algoritmos.

I.3 Operaciones y expresiones en la computadora

Las operaciones y las expresiones son parte fundamental en la especificación de los algoritmos. En esta sección se estudia en qué consisten así como los diversos tipos de operaciones y expresiones.

I.3.1 Operaciones de asignación

Con una operación de asignación se almacena un valor en una variable. El valor se obtiene de una expresión. Cada vez que se asigna un valor a una variable, el valor anterior es eliminado y substituido por el valor asignado.

En un algoritmo, la asignación se representa con un símbolo de igual "=" o con una flecha "←", la estructura de una operación de asignación es la siguiente:

variable ← expresión
o
variable = expresión

Ejemplos de asignación de valores:

`C ← 100`, la variable `C` toma el valor de `100`

`Contador = 0`, la variable `Contador` toma el valor de `0`

`N = 25`, la variable `N` toma el valor de `25`

`Z ← X + 2`, la variable `Z` toma el valor producido por la expresión `X + 2`

`K = K +1`, el valor actual de la variable `K` se incrementa en `1` y el resultado se asigna a la propia variable `K`

`SumaEdad = SumaEdad + Edad`, el valor de la variable `SumaEdad` se incrementa con el valor de `Edad` y el resultado se asigna a la propia variable `SumaEdad`.

`PromedioEdad ← SumaEdad / Contador`, la variable `PromedioEdad` toma el valor producido por la expresión `SumaEdad / Contador`.

I.3.2 Operaciones de entrada y salida de información

Es necesario considerar operaciones que garanticen tanto la entrada de los datos a procesar como la visualización o entrega de los resultados alcanzados. Es decir, se requiere de operaciones de lectura y escritura de información.

En los ejemplos de algoritmos previamente vistos podemos identificar las operaciones de entrada y salida requeridas en cada caso. Como se puede notar en estos ejemplos, una operación de entrada de datos responde al formato:

leer (variable_1, variable_2, …)

mientras que una operación de salida de datos responde al formato:

escribir expresión_1, expresión_2, …

Las operaciones de entrada/salida (lectura/escritura) de datos son parte fundamental de cualquier algoritmo y de un programa de computadora. En el caso de un programa, las operaciones de entrada/salida permiten la interacción del usuario con la aplicación que resulta de su ejecución.

Ejemplos de operaciones de entrada y salida de información:

```
leer (Edad)
leer (Estatura)
leer (base)
leer (altura)
escribir “la edad promedio es: ”, EdadPromedio
escribir “la altura promedio es: ”, AlturaPromedio
escribir “El perímetro es: ”, 2 * (base+altura)
escribir “El área es: ”, base * altura
```

I.3.3 Las expresiones y sus tipos

Las expresiones constituyen uno de los bloques de construcción fundamentales para definir el cuerpo de un algoritmo. Los dos componentes básicos de una expresión son los *operandos* y los *operadores*. Un operando se refiere a las constantes y variables que se involucran en la expresión, y los operadores son símbolos que representan las operaciones que afectan a los operandos, estas operaciones pueden ser aritméticas o lógicas.

I.3.3.1 Expresiones aritméticas

Las expresiones aritméticas son expresiones compuestas que se forman al introducir los operadores aritméticos como parte de la expresión. Los operandos en este tipo de expresión son variables o constantes que corresponden a valores numéricos. Los operadores aritméticos incluyen:

Operador de suma: +
Operador de resta: -

Operador de multiplicación: *
Operador de división: /
Operador de división entera: div
Operación de módulo o resto de la división: mod, %
Operador de exponenciación: ^

Para el lector, seguramente son muy familiares todos los operadores anteriores, excepto el *módulo* o residuo. Daremos una breve explicación del *módulo*. Cuando tenemos $x = A \% y$, se lee: x es *A módulo y,* donde x es el residuo que se produce al hacer una división de A / y. Por ejemplo si $x = 19 \% 4$, entonces $x = 3$, ya que 19/ 4 es 4 y sobran 3. En otras palabras, el operador módulo (mod) produce el residuo de la división entre dos números enteros X y Y (X mod Y), cuando X es divisible entre Y, el módulo es cero.

En la sección I.5.5 se incluyen tablas con los operadores de C/C++ que permiten codificar expresiones aritméticas y lógicas.

Cuando se evalúan operaciones aritméticas dentro de una expresión, el orden de precedencia de las operaciones es el siguiente:

Operaciones encerradas entre paréntesis. Las operaciones encerradas en los paréntesis más anidados son las primeras en ser evaluadas.
Operador exponencial: ^
Operadores de multiplicación y división: *, /
Operadores de división entera y módulo de la división: div, mod
Operadores de suma y resta: +, -

Lo anterior se ilustra en la Tabla I-2.

Operador	**jerarquía**	**operador**
( )	(mayor)	paréntesis
^	↓	potencia
*, / , %	↓	multiplicación, división, modulo
+, -	(menor)	suma, resta

Tabla I-2 Jerarquía de los operadores aritméticos.

Ejemplos de expresiones aritméticas

12 + 8
12 / 5
10 % 5
10 + 3*6
5*(6+14)
25/(6+4)
18*(2 + 3)^2
2*(X + 2)/(Y-2)

2*(X+Y)/(X^2 + Y^2)
X^2 + 2*X*Y + Y^2

I.3.3.2 Expresiones lógicas

Una expresión lógica es aquella que contiene operadores relacionales o lógicos como parte de la expresión. A diferencia de las expresiones aritméticas, las cuales retornan siempre un valor numérico como resultado de su evaluación, las expresiones lógicas devuelven siempre el valor "verdadero" o el valor "falso", como resultado de la evaluación de la operación relacional o lógica contenida en la expresión.

Un operador relacional se utiliza para comparar los valores de dos expresiones, las cuales pueden ser expresiones aritméticas, lógicas, de carácter o de cadena de caracteres. Los operadores relacionales que pueden formar parte de una expresión relacional son los siguientes:

Operador menor que: <
Operador mayor que: >
Operador menor o igual que: <=
Operador mayor o igual que: >=
Operador igual que: ==
Operador diferente de: !=

Los operadores lógicos utilizados en una expresión lógica son los siguientes:

Operador de negación: NOT
Operador de conjunción: AND
Operador de disyunción: OR

Las tablas de verdad de los operadores lógicos NOT, AND y OR se muestran en la Tabla I-3, Tabla I-4 y Tabla I-5, respectivamente.

Expresión	NOT Expresión
Verdadero	Falso
Falso	Verdadero

Tabla I-3 Tabla de verdad del operador NOT.

Expresión 1	Expresión 2	Expresión 1 AND Expresión 2
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

Tabla I-4 Tabla de verdad del operador AND.

Expresión 1	Expresión 2	Expresión 1 OR Expresión 2
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Tabla I-5 Tabla de verdad del operador OR.

Una expresión lógica puede combinar operadores relacionales y operadores lógicos.

Ejemplos de expresiones lógicas:

En la Tabla I-6 se ilustran varios ejemplos de expresiones relacionales y el resultado de su evaluación.

Expresión lógica	Resultado
12 > 8	Verdadero
5 <= 17	Verdadero
25 > 18 + 12	Falso
12 – 4 == 8 + 10	Falso
2*(5 + 4) > (6 + 10)/2	Verdadero
4*(2 + 5)^2 >= 2^2 + 2*5^2	Verdadero
(8 + 12)/4 < 2*(4 + 6)	Verdadero
12 >= 12	Verdadero
18 < 0	Falso
2*(8 + 4) ≠ 24	Falso
(15 <= 24) and (18 >= 14)	Falso
(10 > 4) and (8 < 14)	Verdadero
(15 <= 24) or (18 >= 14)	Verdadero
(17 == 12) or (32 < 18)	Falso
not (64 ≠ 64)	Verdadero
not ((17 == 12) or (32 < 18))	Verdadero
not ((10 > 4) and (8 < 14))	Falso

Tabla I-6 Ejemplos de expresiones lógicas.

I.4 Estructuras de control

Las estructuras de control determinan el flujo de ejecución en un algoritmo y en un programa, es decir, especifican el orden en el que se ejecutan los pasos o instrucciones. En la programación estructurada existen tres tipos fundamentales de estructuras de control:

- Secuencial.
- Selectiva.
- Repetitiva.

En esta sección definimos en que consisten estas tres estructuras y proporcionamos algunos ejemplos. En el capítulo II abordamos cada una de ellas con mayor profundidad aplicando la *metodología por etapas*.

I.4.1 Estructura secuencial

En este tipo de estructura los pasos se ejecutan siguiendo el orden en el que aparecen, es decir, del primer paso en la secuencia al último paso en la secuencia.

Para comprender claramente este tipo de estructura consideremos un fragmento de secuencia genérica dado por:

…, Paso K-1, Paso K, Paso K+1, Paso K+2,…

En dicho fragmento de secuencia, el Paso K+1 no puede ser ejecutado hasta que no se hayan ejecutado primero el Paso K-1 y posteriormente el Paso K. De igual forma, sólo se podrá pasar al Paso K+2 una vez que se haya ejecutado el Paso K+1. Muchos algoritmos propuestos para la solución de problemas simples pueden ser expresados completamente como una estructura secuencial. A continuación veremos algunos ejemplos.

Ejemplo 1.4.- Calcular el resultado de la expresión $x^2 + y^2$.

Algoritmo:

1. Inicio.
2. Leer (x)
3. Leer (y)
4. Xcuad ← x*x
5. Ycuad ← y*y
6. SumaCuad ← Xcuad + Ycuad
7. Escribir "La suma de cuadrados es: " SumaCuad
8. Fin

Diagrama de flujo:

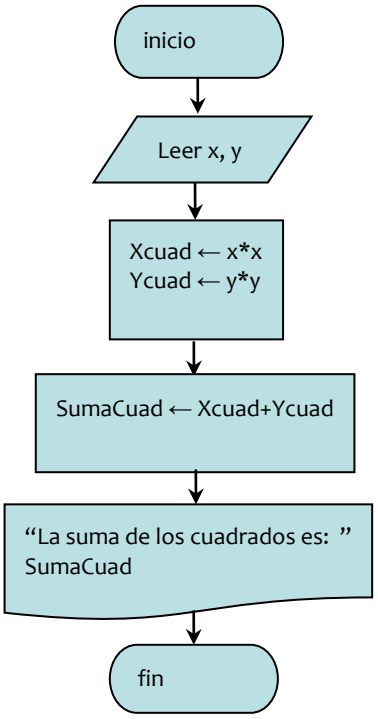


Figura I-6 *Diagrama de flujo para el cálculo de la suma de cuadrados*

Ejemplo 1.5.- Calcular el área y el perímetro de un rectángulo.

Como ya habíamos visto en el Ejemplo 1.1, para el cálculo del área y del perímetro de un rectángulo se requieren como datos de entre la base (b) y la altura (a). La expresión para el cálculo del área es area = base*altura. La expresión para el cálculo del perímetro es perimetro = 2(base+altura). Las salidas que debe producir el algoritmo son el area y el perimetro, como se muestra en el siguiente pesudocódigo.

Pseudocódigo del algoritmo:

```
inicio
   leer (base, altura)
   area ← base*altura
   perimetro ← 2*(base+altura)
   escribir "El área del rectangulo es: " área
   escribir "El perímetro del rectángulo es: " perimetro
fin
```

Diagrama de flujo:

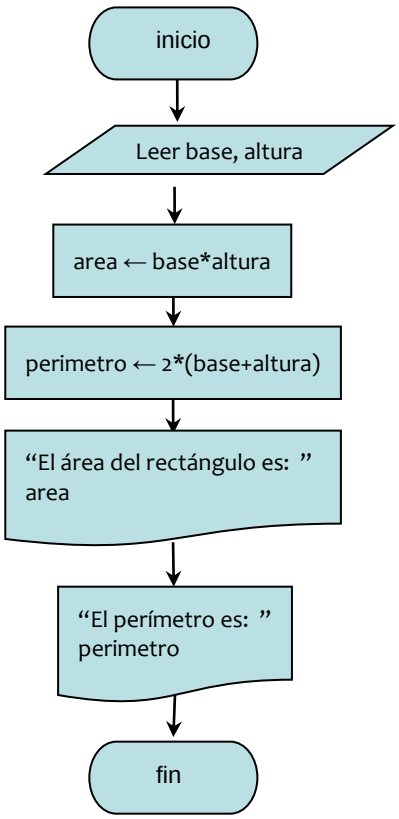


Figura I-7 *Diagrama de flujo para el cálculo del área y del perímetro de un rectángulo.*

I.4.2 Estructuras selectivas

A diferencia de la estructura secuencial, en la cual los pasos del algoritmo se ejecutan siempre en secuencia, las estructuras de selección permiten evaluar una condición lógica y en dependencia del resultado de dicha evaluación se decide cual será el próximo paso a ejecutar. Existen dos tipos de estructura selectiva:

Estructura selectiva sencilla.
Estructura selectiva múltiple.

I.4.2.1 Estructura de selección sencilla sin opción alternativa

Esta estructura permite tomar decisiones en el control del flujo del programa, mediante la evaluación de una condición. En la forma más simple de una estructura de selección sencilla solo se especifica el paso (o grupo de pasos) a ejecutarse si se satisface la condición. La manera de expresar esta estructura con pseudocódigo es con las palabras: **IF-THEN** (en ingles) **SI-ENTONCES** (en español). La Figura I-8 ilustra dos formas de representar un diagrama de flujo con esta estructura.

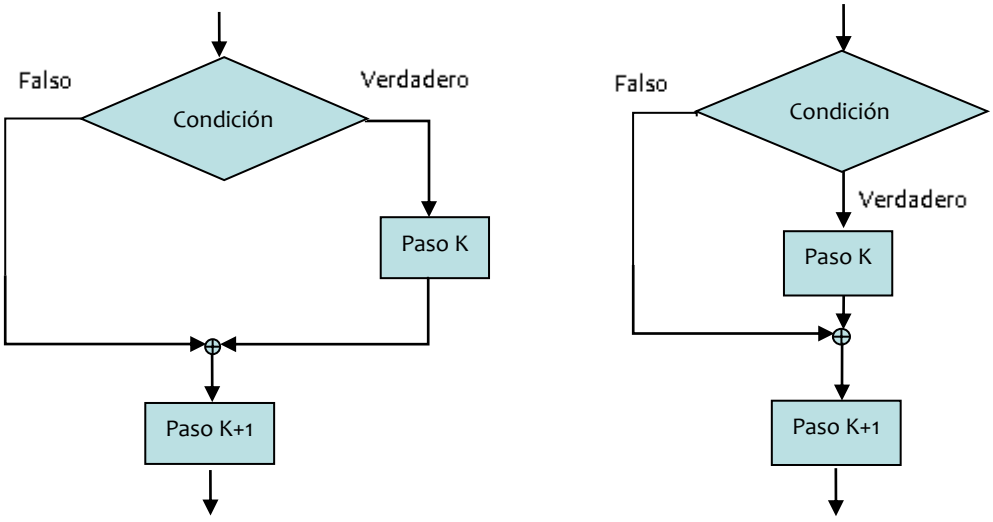


Figura I-8. Representaciones de la estructura de selección sencilla(IF-THEN) en un diagrama de flujo.

Ejemplo 1.6.- La Figura I-9 muestra un ejemplo con la estructura selectiva sin opción alternativa. El diagrama de flujo corresponde a un algoritmo que calcula el porcentaje de alumnos con promedio mayor o igual a 9.0. Como se puede apreciar en dicha figura, si la condición Promedio >= 9.0 es cierta, entonces el contador de promedios altos (ContProm) se incrementa en 1 (ContProm = ContProm +1). Si la condición Promedio >= 9.0 no se cumple, entonces el algoritmo ejecuta el próximo paso (ContTotal = ContTotal +1) sin incrementar el contador de promedios altos. Todo este proceso se repite mientras que el contador sea inferior al numero total de alumnos (NumTotal).

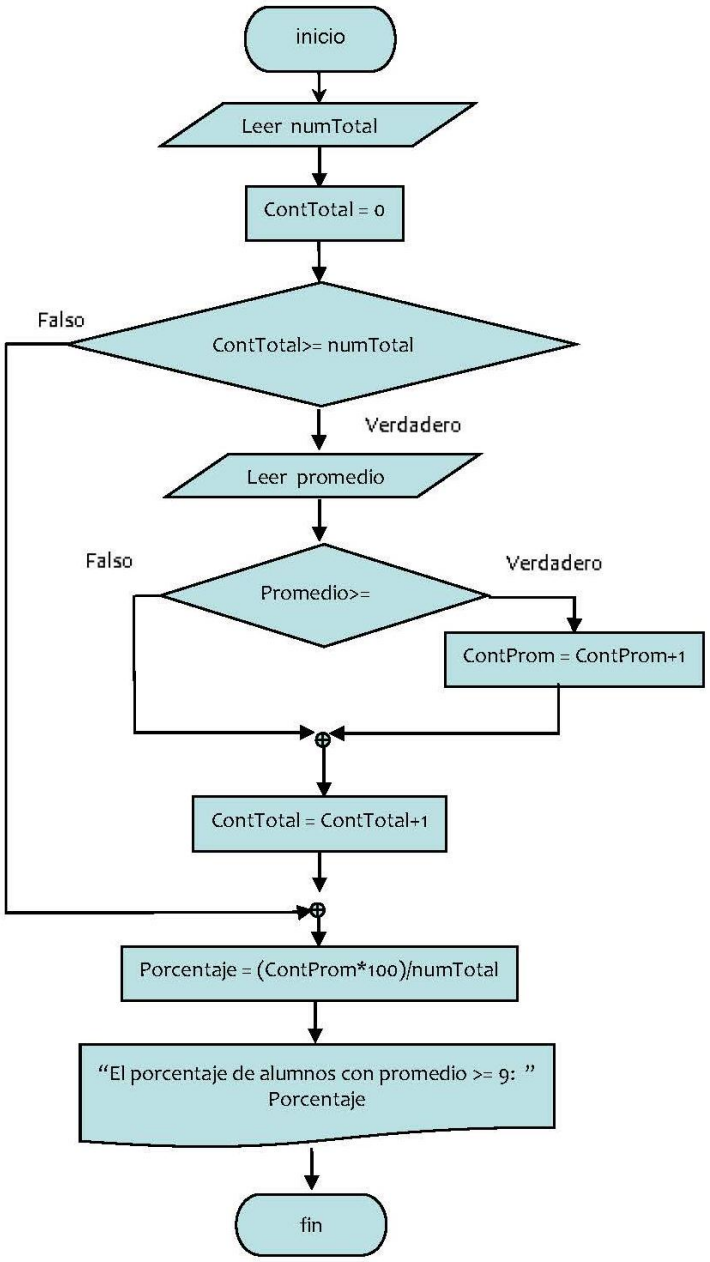


Figura I-9 ***Estructura selectiva sin opción alternativa en un algoritmo que calcula el porcentaje de alumnos con un promedio superior o igual a 9.0.***

I.4.2.2 Estructura de selección sencilla con las dos alternativas

La estructura de selección sencilla completa **IF-THEN-ELSE** (en inglés), **SI-ENTONCES-SINO** (en español), permite seleccionar entre dos alternativas posibles. Es decir, si la condición lógica a evaluar resulta verdadera, entonces se ejecutará una acción o grupo de acciones X_i, en caso contrario se ejecutará otra acción o grupo de acciones Y_i. De esta forma, la estructura de selección completa permite siempre seguir una de dos alternativas en dependencia de la evaluación de la condición lógica. En la Figura I-10 se ilustra el diagrama de flujo de la estructura de selección sencilla con las dos alternativas.

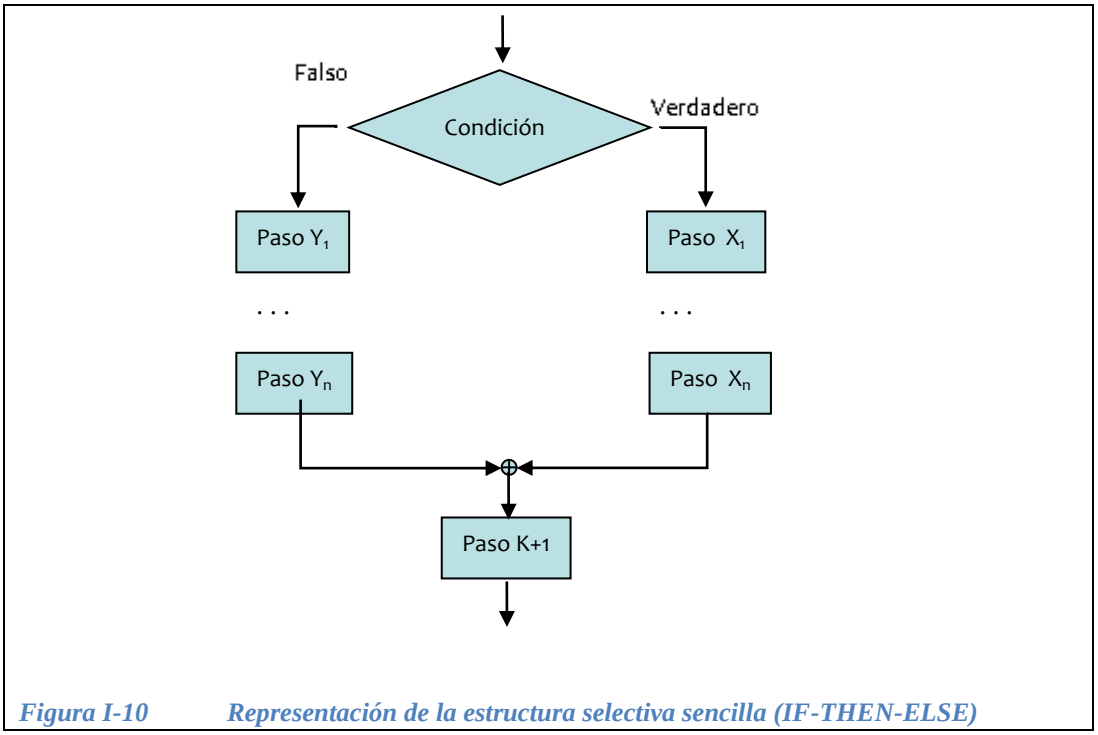


Figura I-10 Representación de la estructura selectiva sencilla (IF-THEN-ELSE)

Ejemplo 1.7.- Calcular la edad promedio según el sexo en un grupo de N deportistas.

Datos de entrada:

Total de deportistas en el grupo (N)
Edad (E)
Sexo de la persona (S)

Datos de salida:

- Edad promedio de deportistas del sexo masculino en el grupo (PM)
- Edad promedio de deportistas del sexo femenino en el grupo (PF)

Operaciones que deben ejecutarse:

- Sumar la edad de las personas del sexo masculino (EM = EM + E)
- Contar la cantidad de personas del sexo masculino (CM = CM + 1)
- Sumar la edad de las personas del sexo femenino (EF = EF + E)

- Contar la cantidad de personas del sexo femenino (CF = CF + 1)
- Calcular la edad promedio para el sexo masculino en el grupo (EPM = (EM/CM)*100)
- Calcular la edad promedio para el sexo femenino en el grupo (EPF = (EF/CF)*100)

Algoritmo:

```
inicio
    Inicializar en cero la cantidad de personas del sexo masculino: CM ← 0
    Inicializar en cero la suma de edad para el sexo masculino: EM ← 0
    Inicializar en cero la cantidad de personas del sexo femenino: CF ← 0
    Inicializar en cero la suma de edad para el sexo masculino: EF ← 0
    Inicializar en cero el total de personas procesadas: TP ←0
    Leer total de deportistas (N)

    mientras TP < N
     Leer sexo de la persona (S)
     Leer edad de la persona (E)
     Si S ='M' ENTONCES
          EM ← EM + E
          CM ← CM +1
          DE LO CONTRARIO
          EF ← EF + E
          CF ← CF + 1
      FinSi
      Incrementar el contador TP en 1: TP ← TP + 1
    fin-mientras

    Calcular la edad promedio para el sexo masculino en el grupo: PM ← (EM/CM)*100
    Calcular la edad promedio para el sexo femenino en el grupo: PF ← (EF/CF)*100
    Escribir "La edad promedio de deportistas del sexo masculino es: ", PM
    Escribir "La edad promedio de deportistas del sexo femenino es: : ", PF
fin
```

Diagrama de flujo: En la Figura I-11 se muestra el diagrama de flujo correspondiente al pseudocódigo anterior.

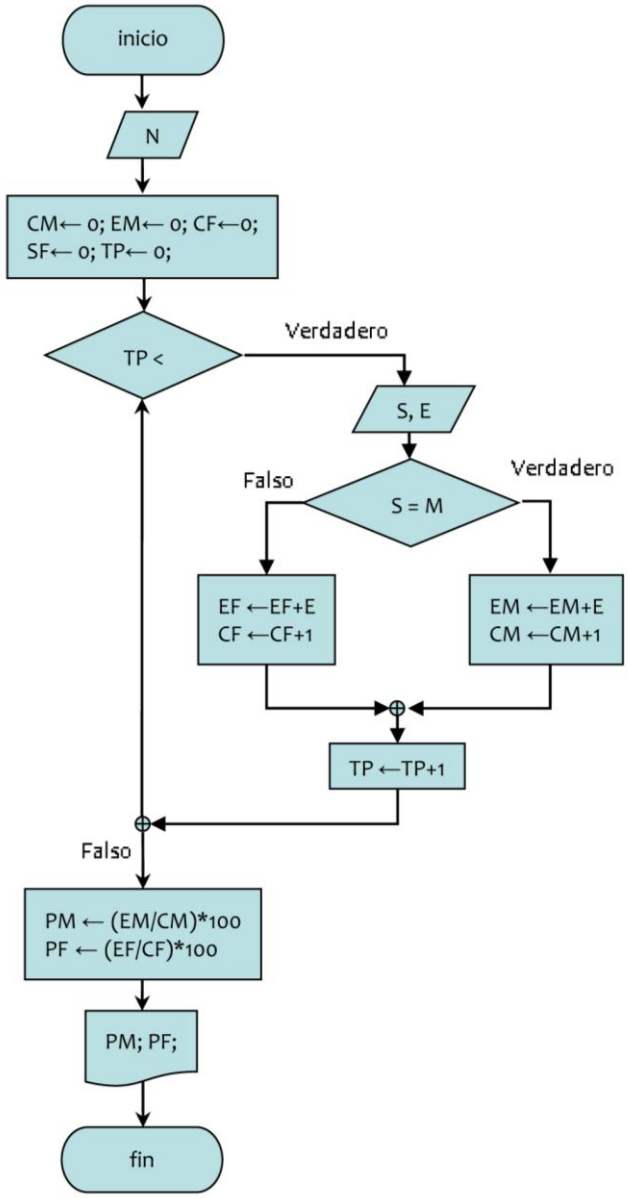


Figura I-11 Representación de la estructura de selección (IF-THEN-ELSE).

I.4.2.3 Estructura de selección múltiple

La selección múltiple se utiliza para seleccionar una de entre múltiples alternativas. Este tipo de estructura se usa cuando el selector puede tomar 1 de N valores. Es decir, a diferencia de la selección simple, en donde solo existen dos alternativas a ejecutar, en la selección múltiple se debe seleccionar una de entre un grupo de N alternativas. Este tipo de problema podría también solucionarse a través del uso de una secuencia o composición de estructuras de selección simple o dobles, sin embargo, esto resulta más confuso que la selección múltiple. Las palabras usadas para representar una selección múltiple en la escritura de un

algoritmo son las siguientes: **en caso de**, para el español; **switch/case**, para el inglés.

En la Figura I-12 se ilustra el uso de la estructura de selección múltiple en dos fragmentos de diagrama de flujo genérico.

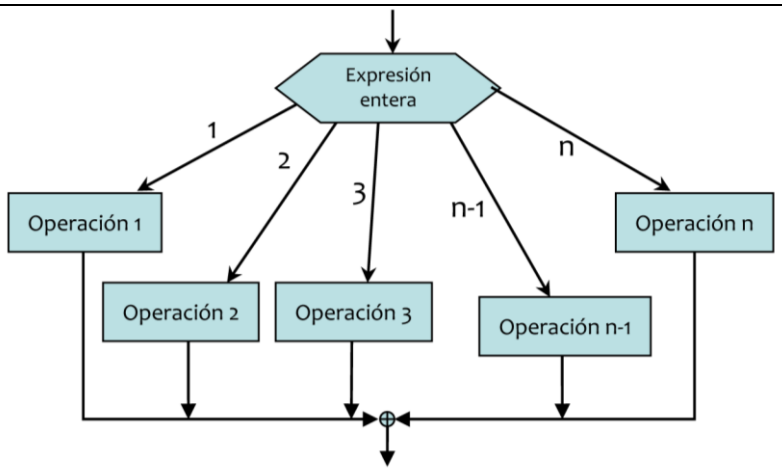


Figura I-12 ***Representación de la estructura de selección múltiple.***

Ejemplo 1.8.- Calcular el peso promedio de un grupo de deportistas de una escuela de perfeccionamiento atlético según la edad, considerando que la edad de los mismos se encuentra en el rango de 16 a 20 años.

Algoritmo:

```
inicio
    Inicializar en cero las variables suma (SP16 ← 0, SP17 ← 0, SP18 ← 0, SP19 ← 0,
                                            SP20 ← 0)
    Inicializar en cero las variables contadoras (TP ← 0, CD16 ← 0, CD17 ← 0,
                                                  CD18 ← 0, CD19 ← 0, CD20 ← 0)
    leer total de deportistas (N)
    mientras TP < N
        leer edad del deportista (E)
        leer peso del deportista (P)
        en-caso-de (E)
                16:     SP16 ← SP16 + P
                        CD16 ← CD16 + 1
                17:     SP17 ← SP17 + P
                        CD17 ← CD17 + 1
                18:     SP18 ← SP18 + P
                        CD18 ← CD18 + 1
                19:     SP19 ← SP19 + P
                        CD19 ← CD19 + 1
                20:     SP20 ← SP20 + P
                        CD20 ← CD20 + 1
        Fin en-caso-de
        Incrementar el contador TP en 1: TP ← TP + 1
    FinMientras

    Calcular el peso promedio para los deportistas de 16 años: PM16 ← (SP16/CD16)*100
    Calcular el peso promedio para los deportistas de 17 años: PM17 ← (SP17/CD17)*100
    Calcular el peso promedio para los deportistas de 18 años: PM18 ← (SP18/CD18)*100
    Calcular el peso promedio para los deportistas de 19 años: PM19 ← (SP19/CD19)*100
    Calcular el peso promedio para los deportistas de 20 años: PM20 ← (SP20/CD20)*100
    Escribir "El peso promedio de los deportistas de 16 años es: ", PM16
    Escribir "El peso promedio de los deportistas de 17 años es: ", PM17
    Escribir "El peso promedio de los deportistas de 18 años es: ", PM18
    Escribir "El peso promedio de los deportistas de 19 años es: ", PM19
    Escribir "El peso promedio de los deportistas de 20 años es: ", PM20
Fin
```

En la Figura I-13 se muestra el diagrama de flujo correspondiente al pseudocódigo anterior.

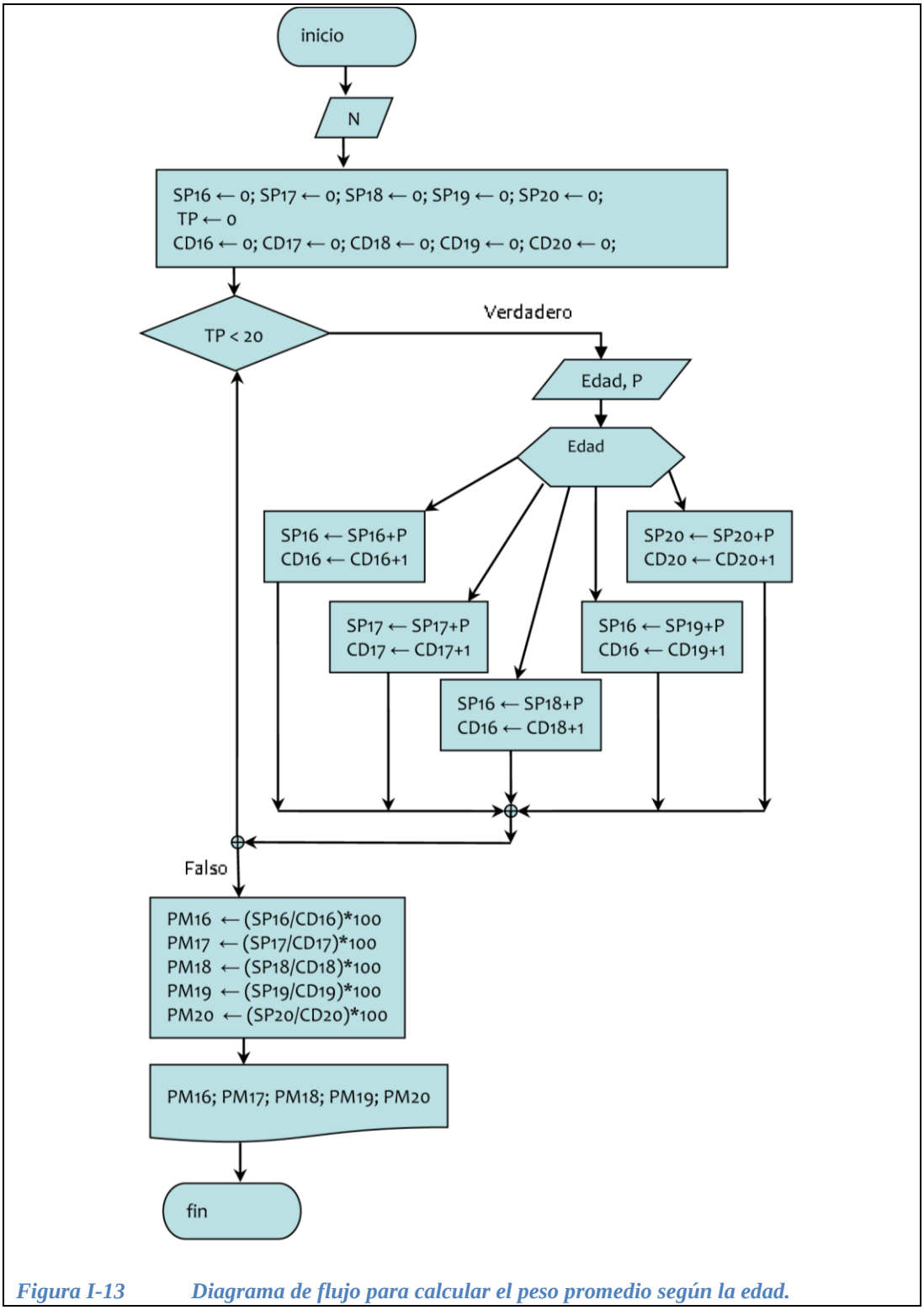


Figura I-13 *Diagrama de flujo para calcular el peso promedio según la edad.*

I.4.3 Estructuras repetitivas

En los ejemplos y problemas de algoritmos y diagramas de flujo hasta aquí presentados, si bien hemos centrado la atención en el uso de las estructuras de selección. En algunos casos ha sido necesario el uso de estructuras de repetición aún cuando no nos hemos referido de forma explícita a las mismas.

Si retomamos el ejemplo 1.1, problema del cálculo del área y el perímetro de un rectángulo y el ejemplo 1.4, problema del cálculo de la expresión $X^2 + Y^2$, notaremos que en ninguno de estos algoritmos fue necesario el uso de estructuras de repetición, ya que en ninguno de estos dos problemas el algoritmo se debe procesar para un conjunto de N datos. Sin embargo, si examinamos los ejemplos 1.2, 1.3, 1.6 y 1.7 notaremos que aquí el uso de estructuras de repetición fue necesario debido a la existencia de un subconjunto de pasos o instrucciones que deben ser ejecutados un número N de veces.

Las estructuras de repetición permiten repetir una operación o bloque de operaciones varias veces. Un bucle o ciclo en un algoritmo o programa es una estructura que repite una sentencia o bloque de sentencias un cierto número de veces. La instrucción (o bloque de instrucciones) que se repite constituye el cuerpo del bucle. Cada repetición del cuerpo del bucle se conoce como iteración del bucle. Las tres estructuras de repetición son:

Estructura repetitiva de control previo (mientras, while).
Estructura repetitiva de control posterior (hacer-mientras, do-while).
Estructura repetitiva de control con contador (desde-hasta, for).

I.4.3.1 Estructura repetitiva de control previo (mientras, while)

La estructura *repetitiva de control previo* (mientras-hacer, while) consta de dos partes:

La condición (expresión lógica) a evaluar.
El cuerpo del bucle, el cual puede ser una sentencia simple o un bloque de sentencias.

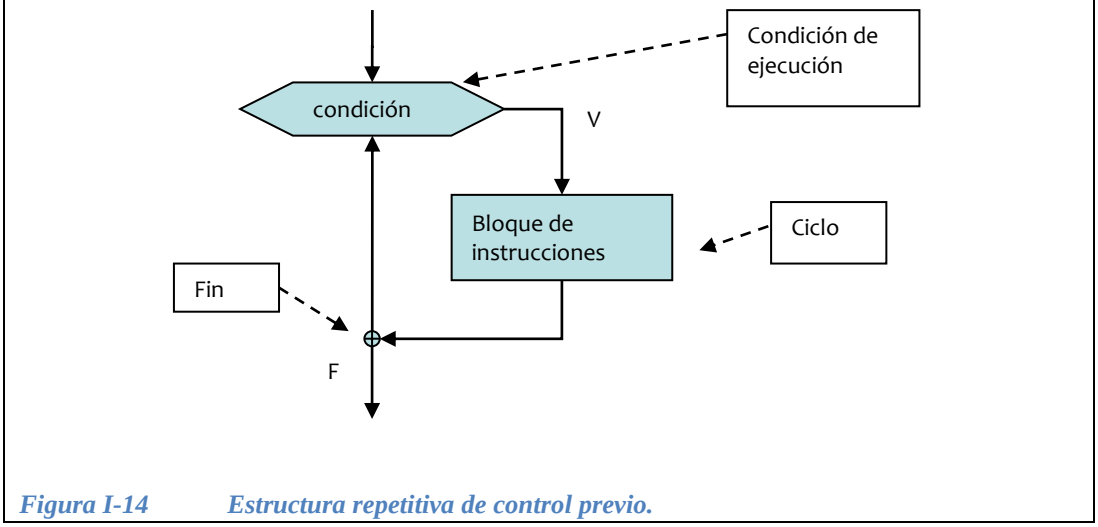


Figura I-14 *Estructura repetitiva de control previo.*

En la estructura *repetitiva de control previo* la condición se encuentra antes del cuerpo del bucle, por lo que necesariamente debe ser verdadera para que el bucle se ejecute al menos una vez, es decir, la condición se evalúa antes de que se ejecute el cuerpo del bucle. El cuerpo del bucle se ejecuta mientras la condición sea verdadera. Cuando la condición deje de ser verdadera, el control se transfiere a la próxima sentencia siguiente a esta estructura. La Figura I-14 muestra el diagrama de flujo correspondiente a la estructura repetitiva *de control previo*.

I.4.3.2 Estructura repetitiva de control posterior (hacer-mientras, do-while)

En la estructura *repetitiva de control posterior* (**hacer-mientras**, **do-while**), la posición de la condición se encuentra después del cuerpo del bucle, por lo que no es necesario que la condición sea verdadera para que el bucle se ejecute al menos una vez. Es decir, la condición se evalúa después que se ejecute el cuerpo del bucle. La estructura *repetitiva de control posterior* **(hacer-mientras**, **do-while**) consta de dos partes:

- El cuerpo del bucle, el cual puede ser una sentencia simple o un bloque de sentencias
- La condición (expresión lógica) a evaluar.

La Figura I-15 muestra el diagrama de flujo correspondiente a la estructura *repetitiva de control posterior.*

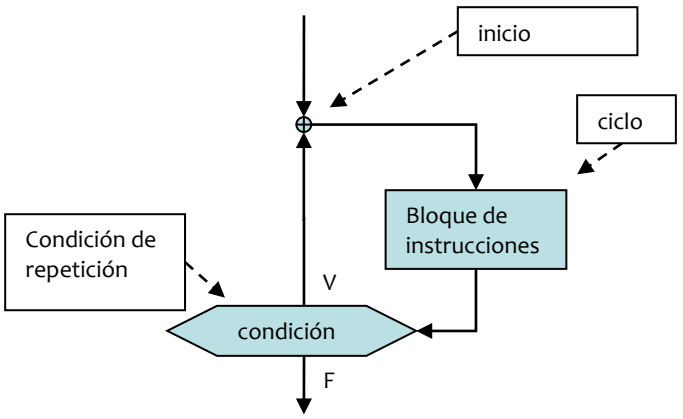


Figura I-15 *Estructura repetitiva de control posterior.*

El ejemplo 1.9 ilustra el uso de la estructura repetitiva de control posterior con una nueva versión del algoritmo y diagrama de flujo para el problema visto en el ejemplo 1.2.

Ejemplo 1.9.- Calcular la edad y peso promedio de un grupo de N estudiantes usando una estructura repetitiva de control posterior.

Datos de entrada:

NumTotal
Edad (para cada estudiante)
Peso (para cada estudiante)

Datos de salida:

EdadPromedio
PesoPromedio

Operaciones a ejecutar:

Pedir el número total de estudiantes: NumTotal
Pedir todas las edades de los estudiantes y calcular la suma: SumaEdades
Pedir todos los pesos de los estudiantes y calcular la suma: SumaPeso
Edad promedio← SumaEdades/NumTotal
Peso promedio← SumaPeso/NumTotal

Algoritmo:

```
inicio
   Leer ( NumTotal)
   i ← 0
   SumaEdades ← 0
   SumaPesos ← 0
   hacer
      leer Edad, Peso
      SumaEdades ← SumaEdades + Edad
      SumaPesos ←SumaPesos + Peso
      i ← i + 1
  mientras i < NumTotal

  EdadPromedio ← SumaEdades/NumTotal
  PesoPromedio ← SumaPesos/NumTotal
  Escribir  "La edad promedio es: ", EdadPromedio
  Escribir  "El peso promedio es: ", PesoPromedio
fin
```

Diagrama de flujo:

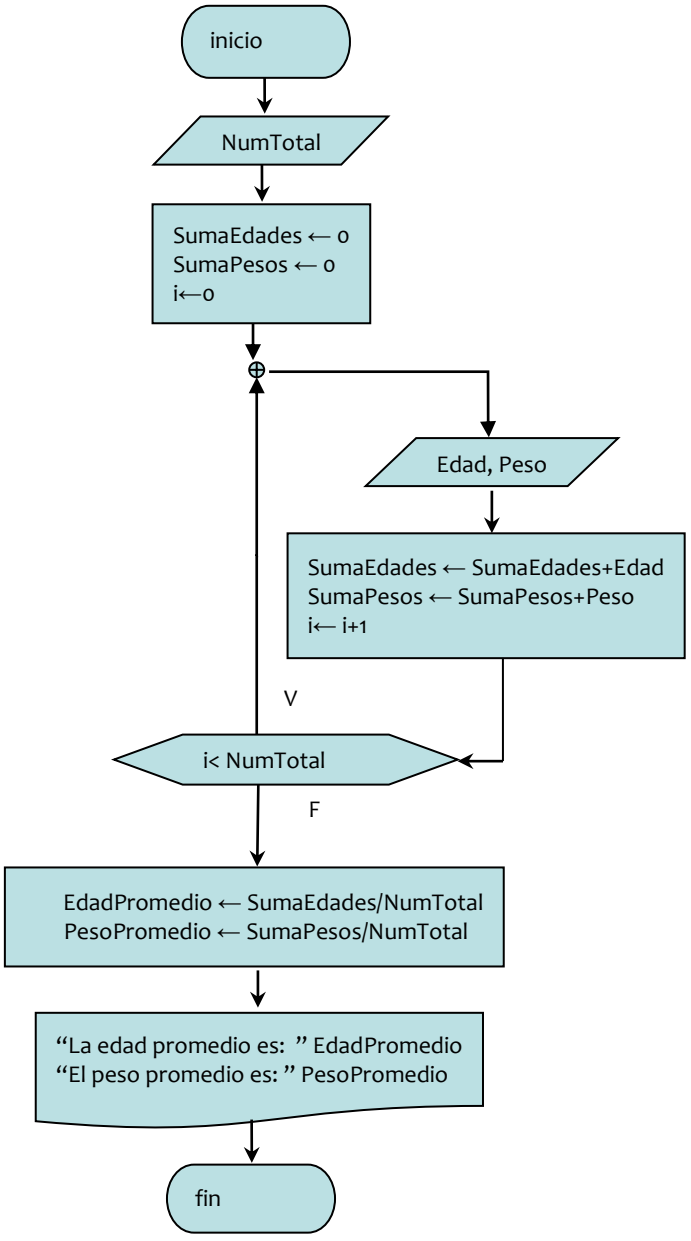


Figura I-16 *Diagrama de flujo para calcular la edad y peso promedio usando una estructura repetitiva de control posterior.*

I.4.3.3 Estructura repetitiva de control con contador (desde-hasta, for)

La estructura *repetitiva de control con contador* **(desde-hasta**, **for**) se utiliza cuando se conoce el número de iteraciones, es decir el número de veces que se ejecutará el bucle. Este tipo de estructura de repetición implementa un bucle controlado por un contador y el bloque de instrucciones se ejecuta una vez por cada valor que toma el contador en un rango especificado de valores. En resumen, el bloque de instrucciones se ejecuta un número fijo de veces.

En una estructura repetitiva de control con contador se pueden identificar cuatro partes:

- Inicialización: se le da un valor inicial al contador.
- Condición: es una expresión lógica. El bloque de instrucciones se ejecuta mientras ésta sea verdadera.
- Incremento: incrementa o decrementa el contador.
- Bloque de intrucciones a iterar.

La Figura I-17 muestra el diagrama de flujo correspondiente a la estructura *repetitiva de control de contador*.

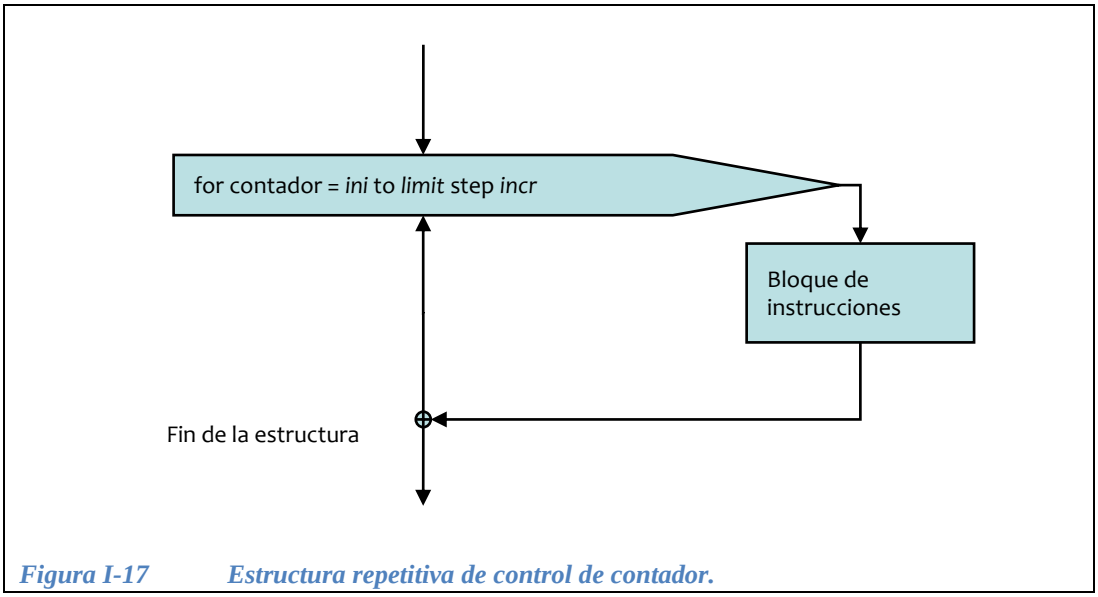


Figura I-17 Estructura repetitiva de control de contador.

Ejemplo 1.10.- Calcular el porcentaje de alumnos con promedio mayor o igual a 9.0 en un grupo de N estudiantes. El valor de N es 50.

Datos de entrada:

- Promedio

Datos de salida:

- Porcentaje de alumnos con promedio mayor o igual a 9.0 (Porcentaje)

Algoritmo:

```
inicio
   K ← 0
   N ← 50
   desde i= 1 hasta N
      leer (promedio)
      si promedio >= 9.0 entonces
            K ←K + 1
       fin-si
       i=i+1
   fin-desde
   porcentaje ← (K*100)/N
   escribir "Porcentaje de alumnos con promedio mayor o igual a 9.0 es: " porcentaje
fin
```

Diagrama de flujo:

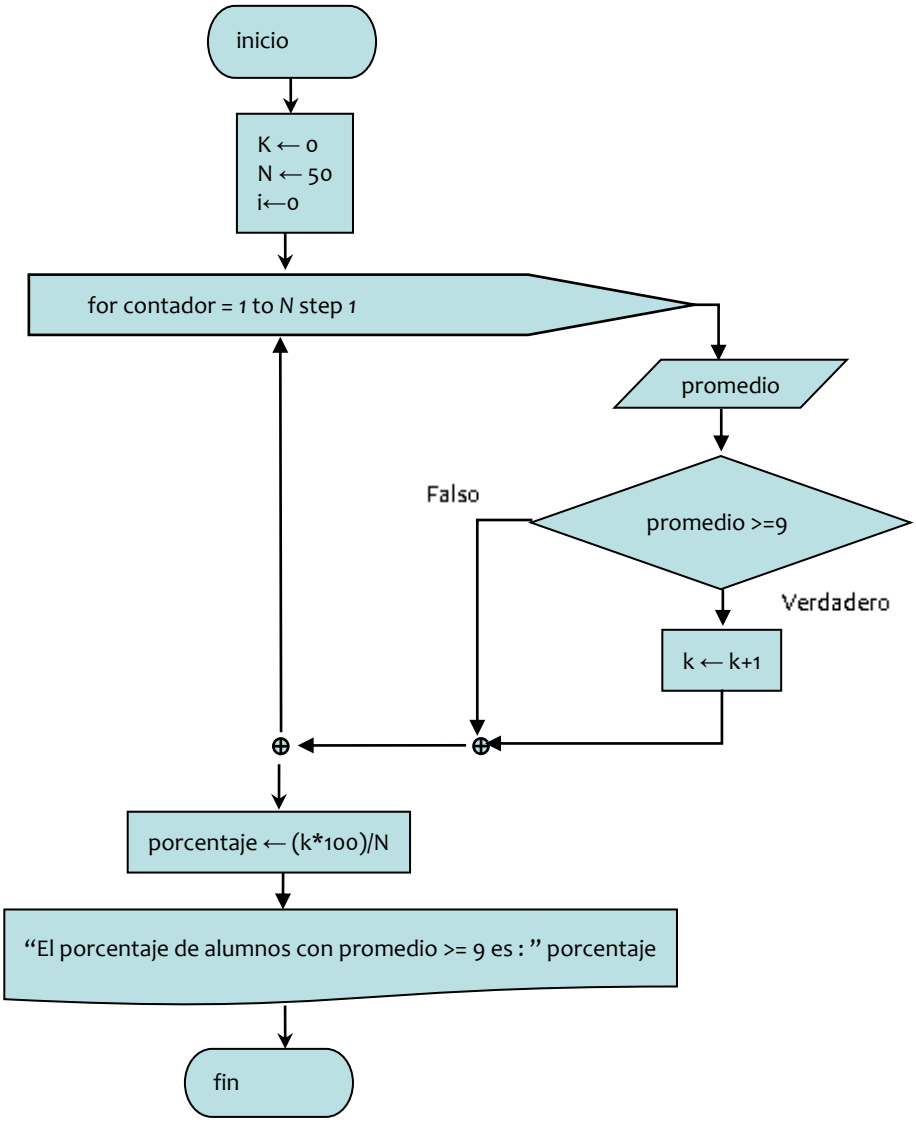


Figura I-18 ***Diagrama de flujo para calcular el porcentaje de alumnos con promedio mayor o igual a 9 en un grupo de N estudiantes usando una estructura repetitiva de control con contador***

I.5 Introducción a C++

I.5.1 Los lenguajes C y C++

El **lenguaje C** es un lenguaje imperativo/procedural, basado en expresiones y funciones. Fue diseñado y desarrollado en el 1972 por D.M. Ritchie de los laboratorios de la AT&T Bell como lenguaje de más alto nivel respecto al lenguaje ensamblador. Kerningham & Ritchie hicieron una definición precisa del lenguaje

C en 1978. Tuvo una gran difusión y en el 1983 se convirtió en estándar ANSI (del inglés, *American National Standards Institute*). El lenguaje C junto con el PASCAL ha sido la base de la ciencia de la computación.

Del C derivaron muchos lenguajes orientados a objetos como C++, Objective-C, C#, y también indirectamente Java, que deriva en primer lugar del C++. En los años 1982-86 Bjarne Stroustrup desarrolló el **lenguaje C++** el cual conserva la eficiencia del lenguaje C pero incorpora el poder de la programación orientada a objetos.

Las principales características del lenguaje C son las siguientes:

- **Flexibilidad**: El C es y ha sido utilizado para escribir una gran variedad de aplicaciones: desde sistemas operativos (OS), a compiladores, a programas de gestión, programas de cálculo científico, video juegos, software embebido, etc.
- **Portabilidad**: El C es en general el lenguaje más disponible para plataformas heterogéneas (Unix, Windows, MSDOS, AIX, Mac, OS, Solaris, etc.).
- **Simplicidad**: El C se basa en pocos conceptos elementales, en particular sobre los conceptos de expresión y función.
- **Eficiencia**: Su naturaleza imperativa/procedural hace que sus formas compiladas sean muy eficientes, por lo que su velocidad de ejecución es elevada.

El lenguaje C++ es un lenguaje orientado a objetos que heredó las ventajas y propiedades del lenguaje C, por lo que es perfectamente posible utilizar el paradigma de la programación estructurada con C++. En este libro usaremos las características de programación estructurada del C++, y dejaremos las características de la programación orientada a objetos para un curso posterior. La ventaja principal del C++ es la simplificación del la interfase de entrada/salida con respecto a C. Con el uso de C++ los alumnos principiantes se concentran en la programación sin tener que invertir mucho tiempo y esfuerzo en el aprendizaje de los comandos de entrada y salida de datos y pueden trabajar fácilmente tanto en pantalla como con archivos.

I.5.2 El preprocesador en C/C++

Cuando una persona escribe un programa en un lenguaje de programación concreto produce lo que se llama *código fuente*. Éste es la "fuente" para generar automáticamente un programa "ejecutable" el cual está escrito en un lenguaje que la computadora entiende y puede ejecutar. A este lenguaje ejecutable se le llama "lenguaje de máquina". El compilador es el programa que convierte el programa fuente en programa ejecutable.

El preprocesador es un instrumento que actúa sobre un programa antes de la compilación: opera sobre un programa fuente y produce un nuevo programa

fuente. En el caso de C/ C++, el preprocesador se invoca automáticamente por el compilador. Las tres tareas principales del preprocesador son:

- Expansión en línea de macros
- Inclusión de archivos (files)
- Compilaciones condicionales

Macros.- Por el término *macro* se entiende la definición de un identificador asociado a una cadena. En C/C++, la macro se define mediante la directiva *#define* y se definen comúnmente al inicio del programa fuente, antes de la función *main* y antes de la definición de las funciones. Por ejemplo, las siguientes macros definen la palabra TRUE con valor de 1 y la palabra FALSE con valor 0:

```
#define TRUE 1
#define FALSE 0
```

Inclusión de archivos.- Es posible incluir en el programa archivos completos mediante la directiva *#include*

Sintaxis:

```
#include "nombre del archivo"
```

o bien:

```
#include <nombre del archivo>
```

El *nombre del archivo* puede ser un nombre de archivo absoluto o relativo, cuando es relativo, el archivo se busca en el directorio corriente. Cuando es absoluto el archivo se busca en el directorio considerado de sistema o de *default.*

Ejemplos comunes:

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

I.5.3 Declaración de variables y de constantes.

Una **variable** es el nombre de una ubicación de almacenamiento en la memoria de acceso aleatorio (RAM: Random Access Memory) de la computadora.

Otra manera de entender una variable, es imaginar que es un cajón de memoria, en donde hay espacio para guardar datos, como un programa requiere de varios cajones, es necesario identificarlos, esto se hace por medio del *"Nombre de la variable"*. También es necesario definir el *tamaño* del cajón, ya que para datos pequeños no es necesario reservar mucha memoria, mientras que para datos más grandes se requiere reservar más memoria (hacer el cajón más grande) el *"tipo de la variable"* esta relacionado con el tamaño del espacio que se reserva en memoria.

Una **constante** es un dato cuyo valor permanece fijo a lo largo de todo un programa, y también requiere de un nombre que la identifique y de un *tamaño* (tipo de dato).

I.5.4 Tipos de datos

El **tipo** de una variable o de una constante indica el grupo de valores que la variable puede guardar (números enteros, de punto flotante, etc.) y el de operaciones que el programa puede hacer con ella (suma, resta, and, or, etc.).

Cuando se crea un programa, se deben declarar las variables indicando al compilador de C++ su tipo y su nombre.

A continuación se enlistan los **tipos básicos** de C/C++.

Enteros con signo :	Rango
`short int`	-128 ... 127
`int`	-32.768 ... 32.767
`long int`	-2.147.483.648 ... 2.147.483.647

Naturales (enteros sin signo):	Rango
`unsigned int`	0 ... 65,535
`unsigned short int`	0 ... 255
`unsigned long int`	0 ... 4,294,967,295

Caracteres:	Rango
`char`	los caracteres ASCII

Punto Flotante:	Rango
`float`	+/-3.402823466 x 10^{38}
`double`	+/-1.7976931348623158 x 10^{308}

Booleanos:	Rango
`bool`	Falso (o cero) o Verdadero (o diferente de cero)

En C/C++ no existe el tipo de dato booleano. Se usan los enteros: Cero (0) para indicar falso y cualquier otro valor entero para indicar verdadero (aunque se sugiere usar uno 1).

Todas las variables deben declararse antes de ser utilizadas. Opcionalmente puede darse un valor de "inicialización". Pueden declararse varias variables del mismo tipo en una misma instrucción.

<tipo> <Nombre Variable> {= <Expresión>|<Valor>};

Ejemplos:

```
float sumaPeso, sumaEstatura;
int edad;
int valorMaximo = 100, valorMinimo = 0;
char sexo;
```

El primer caracter del nombre de una variable debe ser una letra o un guión bajo, no puede empezar con un número.

- Cada caracter se representa internamente con un número, según una codificación estándar.
- La codificación comúnmente usada es el código ASCII, y usa un byte por cada carácter.
- Por lo tanto, se pueden tener 256 caracteres diferentes
- Los primeros 128 caracteres son estándar sobre todos los sistemas
- Los restantes 128 no son soportados siempre por todos los sistemas.
- Van encerrados entre apóstrofes. Por ejemplo, 'm', 'F'
- Caracteres especiales: se usa la diagonal invertida. Ejemplos:

```
newline '\n'          (el cursor baja una línea)
tab '\t'                    (tabulador)
carriage return '\r'     (cambio de línea)
form feed '\f'           (cambio de página)
```

- Cadenas de caracteres

Una cadena es una colección de caracteres delimitada por comillas (" "). En C/C++ las cadenas son secuencias simples de caracteres de los cuales el último, siempre está presente en modo implícito y es '\0'. Por ejemplo:

```
"hasta luego" = {'h', 'a', 's', 't', 'a', ' ', 'l', 'u',
                 'e', 'g', 'o', '\0'}
```

Las cadenas de caracteres van encerradas entre comillas. Ejemplos:

```
"deporte", "carrera", "" (cadena vacía).
```

- El tipo enumerado enum

Crear una enumeración es definir un nuevo tipo de datos, denominado "tipo enumerado" y declarar una variable de este tipo. La sintaxis es la siguiente:

```
enum tipo_enumerado {
    <definición de nombres de constantes enteras>
};
```

Donde tipo_enumerado es un identificador que nombra al nuevo tipo definido.

El siguiente ejemplo declara una variable llamada `color` del tipo enumerado `colores`, la cual puede tomar cualquier valor especificado en la lista:

```
enum  colores{ azul, amarillo, rojo, verde, blanco, negro };

colores color;

color = azul;
```

Más adelante, en la sección II.4 del capítulo II se estudia con más detalle el tipo `enum`.

I.5.5 Operadores matemáticos, relacionales y lógicos

A continuación, se incluyen tablas con los operadores de C/C++ que permiten codificar expresiones aritméticas y lógicas.

Operadores matemáticos básicos:

Operador	Significado	Ejemplo
=	Asignación de un valor	(calificación = 10)
+	Suma	(x = 8 + y)
-	Resta	(x = 8 - y)
*	Multiplicación	(x = 8 * y)
/	División	(x = 8 * y)
%	Módulo o residuo	(x = 8 % y)

Operadores matemáticos adicionales:

Operador	Significado	Ejemplo
++	Incremento en 1	(i++ es equivalente a i = i +1)
--	Decremento en 1	(i- - es equivalente a i = i-1)
+=	Adición compuesta	(x += 8 equivale a x = x + 8)
-=	Sustracción compuesta	(x -= 8 equivale a x = x - 8)
*=	Multiplicación compuesta	(x *= 8 equivale a x = x * 8)
/=	División compuesta	(x /= 8 equivale a x = x / 8)

Operadores relacionales:

Operador	Prueba	Ejemplo
= =	Si dos valores son iguales:	(calificación = = 10)
!=	Si dos valores son diferentes	(viejo != nuevo)
>	Si el primer valor es mayor que el segundo	(precio > 300)
<	Si el primer valor es menor que el segundo	(sueldo < 5000)
>=	Si el primer valor es mayor o igual al segundo	(promedio>=8)
<=	Si el primer valor es menor o igual al segundo	(edad<=17)

** Error: Es muy común confundir el operador relacional "= = "con el operador de asignación " = " de la siguiente manera:

`if (a = b)` El compilador NO envía un mensaje de error y el resultado es que se hace la asignación de `b` en `a` y el valor de la expresión lógica depende del valor asignado a `a` (si es cero resulta FALSO y si no, resulta VERDADERO).

```
if (a == b)
```
Manera correcta de expresar “si a igual con b”

Operadores lógicos:

Operador	Operación lógica	Ejemplo
&&	AND	(x && y equivale a: x AND y)
||	OR	(x || y equivale a: x OR y)
!	NOT	(!x equivale a NOT x)

Los operadores relacionales y lógicos producen resultados “verdadero” y “falso”. En C y C++ un resultado “verdadero” puede ser cualquier valor diferente de cero y un resultado “falso” es siempre cero.

I.5.6 Estructura básica de un programa en C/C++

Un programa en C/C++ consta normalmente de cuatro partes principales

Inclusión de archivos.
Declaración de variables globales.
Declaración de funciones y subrutinas.
El programa principal.

Las funciones y subrutinas se estudiarán en el capítulo V.

Antes de comenzar a escribir el código de un programa, es conveniente poner un comentario en las primeras líneas que describa a grandes rasgos lo que hace el programa e indicar el nombre del archivo en el que se guarda dicho programa, por ejemplo, en el caso de los archivos en C++: *nombre*.cpp.

A continuación se presenta un esqueleto con la estructura básica de un programa en C/C++:

```
// comentarios sobre lo que hace el programa y nombre del
// archivo
//  nombre.cpp
#include <iostream.h>
#include <math.h>
       ::

[<declaraciones-de-variables-globales>]

[<declaraciones-de-funciones-y-subrutinas>]

main( ) {

  [<declaraciones-y-definiciones-de-variables-locales>]

  [<secuencia de instrucciones>]

  return 0;
}
```

Las primeras líneas son de comentarios y por lo tanto no son obligatorias, sin embargo se recomiendan porque es muy útil documentar el programa. A continuación se incluyen los archivos de encabezado (los mostrados aquí son solo ejemplos). Las declaraciones y definiciones pueden ir antes del programa principal (`main`) o bien dentro del programa principal. La manera conveniente de hacerlo se estudia en el capítulo V "Fundamentos de diseño modular" con los conceptos de variable local y global.

I.5.6.1 Archivos de encabezado

Los archivos con extensión *.h que se incluyen al comienzo de un programa se llaman “archivos de encabezado”.

Normalmente estos archivos se encuentran en el subdirectorio INCLUDE y contienen definiciones que el compilador proporciona para diferentes operaciones.

Por ejemplo, hay un archivo de encabezado que brinda definiciones para operaciones matemáticas (<math.h>) otro para operaciones de entrada y salida (<iostream.h>) otro para formato de entrada y salida (<iomanip.h>), etc.

Para incluir uno de estos archivos debe hacerse al principio del programa y con el siguiente formato:

```
#include <nombrearchivo.h>
```

por ejemplo:

```
#include <iostream.h>
```

I.5.6.2 El programa principal

La sentencia `main()` identifica al programa principal. Éste es el punto de inicio en la ejecución del programa en la computadora. Un programa solo puede tener una función llamada `main`.

La parte <`main`> es la única que es obligatoria en todo programa escrito en C o C++.

I.5.6.3 Recomendaciones para hacer un programa legible

Los buenos programas deben ser fáciles de leer y entender. A continuación se listan cuatro recomendaciones para hacer un programa legible:

1. Usar nombres de variables que tengan significado, es decir, que describan su función. Por ejemplo:

   ```
   int x, y, z;
   ```

   No da información acerca de la función de las tres variables, la siguiente declaración queda mucho más clara:

   ```
   int edad_alumno, calificacion, peso;
   ```

2. Respetar las alineaciones y sangrías apropiadas para cada instrucción.
3. Dejar líneas en blanco para separar instrucciones no relacionadas.
4. Emplear comentarios que expliquen el procesamiento del programa. Aquí cabe señalar que deben evitarse los comentarios que no agregan información al código, por ejemplo el comentario:

   ```
   x = 3.5;      // se asigna 3.5 a la variable x
   ```

   es completamente innecesario.

I.5.6.4 Reglas de codificación.

Si bien las instrucciones en C/C++ para las diferentes estructuras de control se estudian en el capítulo II, adelantamos aquí un breve conjunto de reglas para hacer la codificación más legible. Es recomendable referirse a estas reglas conforme se vaya avanzando en los temas para que los términos usados resulten familiares.

1) Control de flujo

a) Las instrucciones que son controladas por instrucciones de control de flujo deben escribirse en líneas separadas de la instrucción que las controla.

2) Indentación

a) Los bloques de instrucciones deben estar indentados.
b) Las instrucciones de control de flujo llevan siempre indentación en las instrucciones que controlan. **if, for, while, do/while, switch**.

3) Espacio en blanco

a) Las instrucciones que están relacionadas deben distinguirse de las que no lo están mediante una línea en blanco entre ellas.

b) Nunca debe haber una línea en blanco seguida de otra línea en blanco.

4) **Comentarios**
   a) Se deben insertar comentarios que aclaren el propósito de las instrucciones o de grupos de instrucciones.
   b) Los comentarios deben tener la misma indentación que las instrucciones a las que se refieren.
5) **Nombres de las variables**
   a) Los nombres de cada variable deben indicar claramente el significado del dato que contienen.

I.6 Resolución de un problema mediante un programa

Un buen programador debe seguir un proceso de diseño. Dentro de la fase de *diseño del algoritmo* que soluciona el problema es recomendable estructurar el algoritmo siguiendo la estrategia de *arriba-abajo*, con esto se definen los principales pasos a seguir.

Cada uno de los pasos del problema debe dividirse en varios "subproblemas", hasta que se llegue a funciones que cumplan con una sola tarea específica. A esto se le llama la estrategia de "*divide y vencerás"*. En el "capítulo V: Fundamentos de Diseño Modular" se expone este tema con detalle. En la Figura I-19 se ilustran las fases de resolución de un problema mediante un programa. El primer paso consiste en la descripción y análisis del problema, este paso permite comprender de qué se trata el problema a resolver. En el segundo paso se diseña el algoritmo que resuelve el problema. Posteriormente se implementa el algoritmo mediante código, a este paso se le llama codificación. Una vez que el código esta libre de errores, se ejecuta el programa y, finalmente, la validación consiste en verificar que dados unos datos de entrada se producen las salidas deseadas.

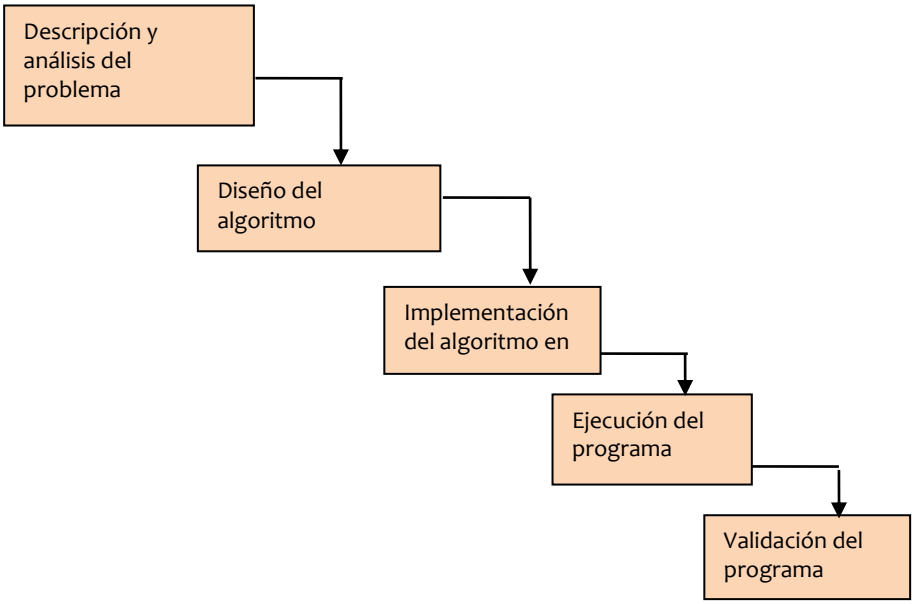


Figura I-19 *Fases de resolución de un problema mediante un programa.*

I.6.1 Construcción de un programa

El proceso de construcción de un programa corresponde a la fase denominada *implementación*. La fase de *edición* es la primera etapa de la construcción del programa. Durante ésta, se escribe el *programa fuente*. Una vez que el programa está editado sigue la fase de *compilación* la cual permite convertir un programa fuente a un programa objeto. Lo normal es que aparezcan errores de compilación la primera vez que se compila un programa fuente. Una vez que éstos se arreglan, la fase de *ligado* (linking) convierte un programa objeto a programa ejecutable. El programa ejecutable se ejecuta directamente sobre el sistema operativo. Durante la fase de *depuración* (debugging) se da seguimiento a la ejecución del programa paso a paso en busca del origen de alguna falla en los resultados esperados. La secuencia de los pasos de la construcción de un programa se ilustran en la Figura I-20.

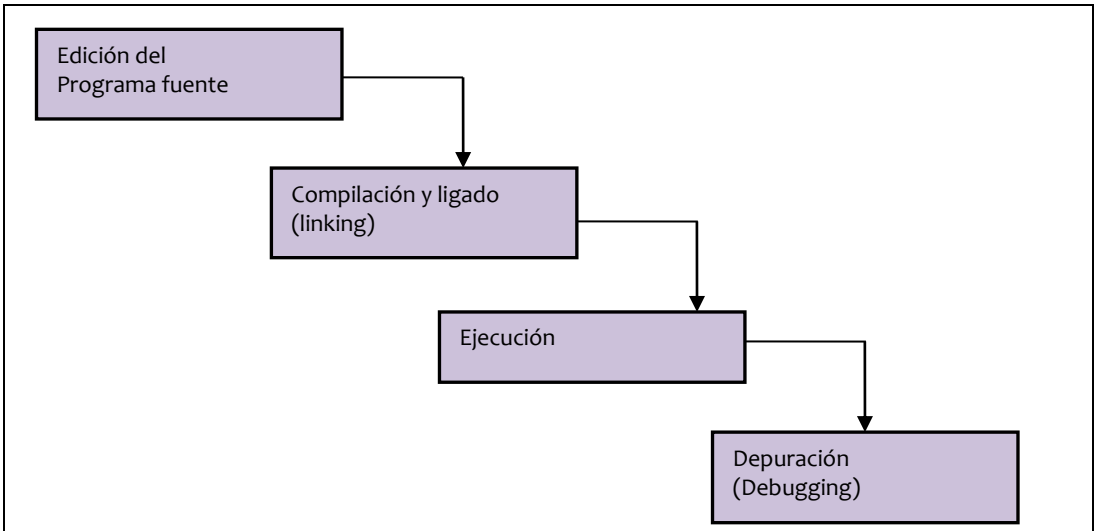


Figura I-20 Fases de construcción de un programa.

I.6.2 Operaciones de entrada y salida de datos

Los programas hacen uso de datos externos que deben entrar al programa para procesarlos y producir con estos los resultados deseados en forma de datos de salida. Los datos de entrada pueden provenir del teclado o de algun otro dispositivo como el disco duro o el mouse, por ejemplo. Los datos de salida pueden tener como destino la pantalla de la computadora o también otros dispositivos como el disco duro entre otros.

I.6.2.1 Teclado y pantalla.

Para poder comunicarnos con la computadora como usuarios es necesario introducir datos desde el teclado y desplegar resultados en pantalla. Para hacer esto existen dos tipos de operaciones:

Entrada desde teclado: Se toma la información del teclado y se asigna a una variable del programa de la siguiente manera

```
x ← teclado
y ← teclado
```

Salida a pantalla: Se envían los datos producidos hacia la pantalla como si se tratara de asignar un valor a una variable del programa de la siguiente manera

```
pantalla ← x + y
pantalla ← x - y
```

La finalidad de trabajar con el lenguaje C++ en este libro, es simplificar el uso de las instrucciones de entrada y salida. Trabajaremos con los objetos `cin` y `cout` para leer datos del teclado y escribir datos en pantalla. Como este curso no incluye

la programación orientada a objetos, omitiremos la explicación detallada sobre lo que son estos objetos y nos limitaremos a utilizarlos.

`cin` sirve para capturar datos del teclado. Utiliza el operador >> que indica la dirección en la que fluyen los datos. En el siguiente ejemplo el dato que se escribe con el teclado fluye desde el teclado hacia la variable x en el programa.

```
cin >> x;
```

`cout` es la instrucción que utilizaremos para escribir datos en pantalla. Utiliza el símbolo << que indica la dirección en la que fluyen los datos. El siguiente ejemplo indica que el resultado de la expresión `x+y` será enviado hacia la pantalla.

```
cout << x+y;
```

A los símbolos >> y << se les llama *insertores*, para recordar fácilmente la dirección adecuada de estos símbolos, recomendamos suponer que las variables del programa siempre están del lado derecho, así como se ilustra en la Figura I-21, de tal forma que la instrucción de entrada va de izquierda a derecha (teclado al CPU) y la de salida va de derecha a izquierda (CPU a la pantalla):

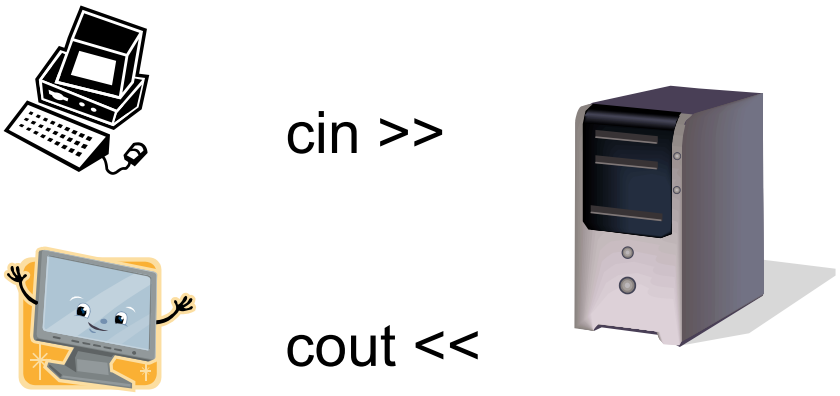


Figura I-21 ***Entrada desde el teclado hacia el CPU y salida desde el CPU hacia la pantalla***

Después de `cin >>` debe ponerse el nombre de la variable en donde se va a capturar el dato. De esta manera el valor que el usuario introduzca a través del teclado es el que se guardará en la variable:

```
cin >> <variable>;
```

Para desplegar en pantalla el resultado de la expresión, usamos:

```
cout << <expresión>;
```

Con la instrucción `cout` también es posible desplegar mensajes en la pantalla, el mensaje debe estar encerrado entre comillas:

```
cout << "Hola!";
```

A continuación presentamos el primer ejemplo de programa en C++, es el código de un programa que envia un mensaje de saludo al usuario.

```
#include <iostream>
int main( )
{
    cout << "Hola mundo! ";
    return 0;
}
```

Otro ejemplo.- Lo que hace el siguiente programa es pedir la edad del usuario y dar una recomendación en función de si es mayor de edad o no.

```
// pregunta tu edad  edad.cpp
#include<iostream.h>

int edad;

main( ) {

  cout << "que edad tienes?";
  cin >> edad;
  if( edad <= 17 )
    cout << "tomate un refresquito";
  else
    cout << "¿Qué te tomas?";

  return 0;
}
```

I.6.2.2 Archivos en disco

Para tener disponibles los datos entre las diferentes ejecuciones de un programa, es necesario almacenarlos en memoria permanente. Un archivo es un fragmento de memoria que proporciona el sistema operativo, con el fin de guardar la información en memoria permanente (disco duro, CD, memory stick, etc.)

Para poder utilizar un archivo es necesario **abrirlo**. Una vez que se ha terminado de trabajar con el archivo ya sea para lectura o para escritura, es necesario **cerrarlo**.

Cuando trabajamos archivos con C++, utilizamos *identificadores*. Estos identificadores pueden corresponder con dispositivos de entrada o de salida. Con estos identificadores podemos realizar las cuatro operaciones que se hacen con archivos, que son:

- Abrir un archivo.
- Leer de un archivo.
- Escribir a un archivo.
- Cerrar un archivo.

Si la operación que se requiere es de *lectura*, entonces el identificador se declara de la siguiente forma:

```
ifstream identificador; // para entrada de datos
```

Y si la operación que se requiere es de *escritura*, entonces declararemos al identificador como:

```
ofstream identificador; // para salida de datos
```

Apertura de un archivo:

Además del *identificador* que se utiliza para leer o escribir datos en un archivo, es necesario proporcionar el nombre del archivo que se necesita abrir, en C++ un archivo se abre con la siguiente instrucción:

```
identificador.open("nombreArchivo.txt");
```

Lectura de un archivo:

Cuando se leen datos de un archivo, éstos se toman en el orden en el que están en guardados. El archivo es visto como un "torrente" de datos o *stream*.

```
identificadorEntrada  >>  variableDestino;
```

Escritura en un archivo:

Cuando escribimos en un archivo, los datos enviados se van acumulando y no son substituidos como en una asignación, es decir, se va formando un "torrente" de datos o *stream*.

```
identificadorSalida  <<  variableFuente;
```

Cierre de un archivo:

Para cerrar un archivo, se requiere el *identificador* que se utilizó para leer o escribir los datos y proporcionar el nombre del archivo que se desea cerrar:

```
identificador.close("nombreArchivo.txt");
```

Ejemplo 1.11.- Hacer un programa que lea un número de un archivo de texto llamado "datos.txt" y que escriba su cuadrado y su cubo en otro archivo de texto llamado "resultados.txt".

```
//Programa que obtiene el cuadrado y el cubo de un número
# include <fstream.h>
# include <math.h>

int numero, cuadrado, cubo;
Ifstream entrada;
ofstream salida;

main ( ) {

  entrada.open("datos.txt");
  entrada >> numero;
  entrada.close();

  cuadrado = numero * numero;
  cubo = pow( numero, 3 );

  salida.open("resultados.txt");
  salida << "El cuadrado del número es " << cuadrado << endl;
  salida << "El cubo es " << cubo;
  salida.close();

  return 0;
}
```

El archivo "datos.txt" ya debe existir y contener los datos que se van a leer. De lo contrario habrá problemas.

Si el archivo "resultados.txt" no existe, será creado y luego abierto. Si ya existe será borrado y luego abierto.

Ejemplo I.12.- Hacer un programa que lea de un archivo de texto la medida de la base y la altura de un rectángulo y que escriba en otro archivo de texto el perímetro y el área de tal rectángulo.

```
//Programa que obtiene el perímetro y la superficie de un
rectángulo rectangulo.cpp//
# include <fstream.h>
# include <math.h>
int base, altura, perimetro, area;

Ifstream datos;
ofstream resultados;

main ( ) {

  datos.open("datosRectangulo.txt");
  datos >> base;
  datos>>altura;
  datos.close();

  perimetro = 2 * ( base + altura );
  area = base * altura;

  resultados.open("resultRectangulo.txt");
  resultados << "El perimetro del rectangulo es "
             << perimetro << endl;
  resultados << "El area es " << area;
  resultados.close();

  return 0;
}
```

Capítulo II Estructuras de control de flujo

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

Diseñar diagramas de flujo y programas en C++ que contengan:

- Estructura secuencial
- Estructura selectiva
- Estructura repetitiva

II.1 Estructura secuencial

La estructura secuencial es la más simple de las estructuras de programación, en ella se ejecutan una serie de pasos siguiendo el orden de una secuencia, es decir, primero se ejecuta una operación, después otra, después la que sigue, y así sucesivamente. En general, los programas más simples tienen la estructura secuencial que se muestra en la Figura II-1.

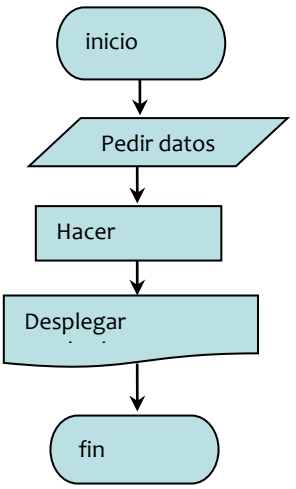


Figura II-1 *La estructura secuencial de los programas más simples.*

La sección *hacer cálculos* de la Figura II-1 puede consistir en un solo cálculo sencillo, o en una serie de cálculos complicados. Lo mejor es detallar el diagrama de flujo de tal forma que muestre con claridad cómo funciona el algoritmo y para qué sirve. El problema resuelto del siguiente ejemplo tiene estructura secuencial.

Ejemplo 2.1.- Obtener el cuadrado y el cubo de un número.

Primero es necesario *definir las entradas*, en este caso la entrada es una sola y le llamaremos `numero`. Después hay que *definir las salidas*, pondremos los resultados en las salidas: `cuadrado` y `cubo`. La descripción de cómo se resuelve este problema queda definida con el algoritmo de la Figura II-2.

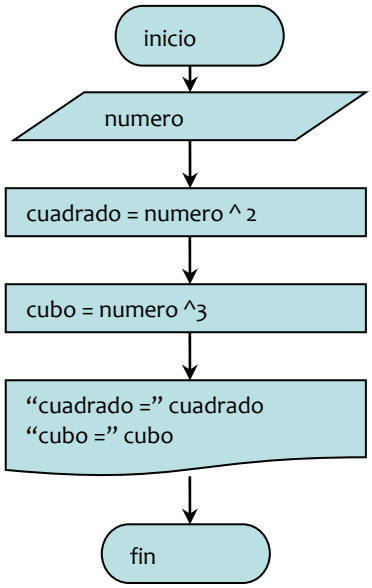


Figura II-2 *Obtención del cuadrado y del cubo de un número.*

A continuación se muestra el código en C++ para resolver el problema del ejemplo anterior.

```
// Programa que obtiene el cuadrado y el cubo de un número
// potencia.cpp (nombre del archivo que contiene este programa)

#include <iostream.h>
#include <math.h>

main() {
  // Declaración de las variables
  int numero, cuadrado, cubo;

  // Pedir los datos
  cout << "Cual es el número? ";
  cin >> numero;

  // Hacer los cálculos
  cuadrado = numero * numero;
  cubo = pow( numero, 3 );

  // Desplegar los resultados
  cout << "El cuadrado del número es " << cuadrado << endl;
  cout << "El cubo es " << cubo;

  return 0;
}
```

Contiene lo necesario para operaciones de entrada/salida

Para usar funciones de la biblioteca matemática, como `pow()` y `sqrt()`

Sirve para poner lo que sigue en otro renglón

Ejemplo 2.2.- Construir un diagrama de flujo tal que, dados como datos la base y la altura de un rectángulo, calcule el perímetro y la superficie del mismo.

Definiendo entradas: en este caso, las entradas son `base` y `altura`.

Definiendo salidas: les llamaremos `perimetro` y `area`.

En la Figura II-3, se muestra la solución del algoritmo, solo falta detallar los cálculos.

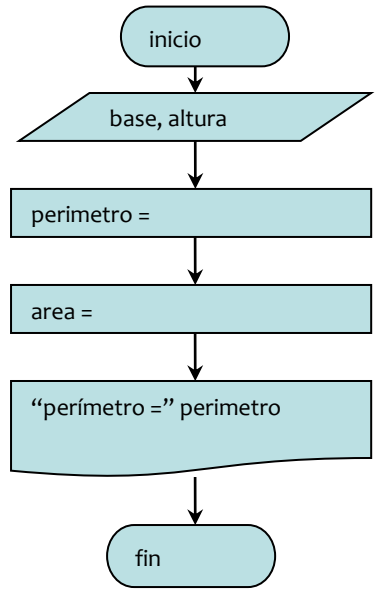


Figura II-3 ***Obtención del área y el perímetro de un rectángulo, solución parcial.***

El diagrama completo de este algoritmo se muestra en la Figura II-4.

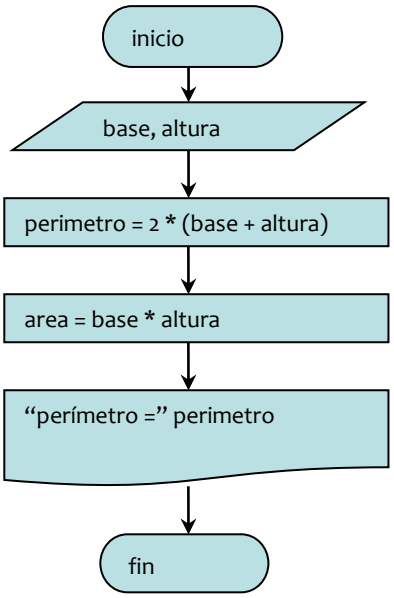


Figura II-4 *Obtención del área y el perímetro de un rectángulo, solución completa.*

¿Cómo completaríamos el siguiente programa que resuelve el problema del ejemplo anterior?

```
// Obtiene el perímetro y el área de un rectángulo
// rectangulo.cpp (nombre del archivo que contiene este programa)

#include <iostream.h>

main() {
  // Declaración de las variables
  int base, altura, perimetro, area;

  // Pedir los datos
  cout << "¿Cual es la base? ";
  cin >> base;
  cout << "¿Cual es la altura? ";
  cin >> altura;

  // Hacer los cálculos

  // Desplegar los resultados
  cout << "El perímetro es " << perimetro << endl;
  cout << "El área es " << area;

  return 0;
}
```

Aquí hacen falta las operaciones para obtener el área y el perímetro

A continuación se muestra el programa completo que resuelve este ejemplo.

```
// Obtiene el perímetro y el área de un rectángulo
// rectangulo.cpp

#include <iostream.h>

main() {
  // Declaración de las variables
  int base, altura, perimetro, area;

  // Pedir los datos
  cout << "¿Cual es la base? ";
  cin >> base;
  cout << "¿Cual es la altura? ";
  cin >> altura;

  // Hacer los cálculos
  perimetro = 2 * ( base + altura );
  area = base * altura;

  // Desplegar los resultados
  cout << "El perímetro es " << perimetro << endl
       << "El área es " << area;

  return 0;
}
```

Mientras no pongamos punto y coma se puede usar un solo `cout`

Otra opción es la siguiente, observar que con esta opción ahorramos el uso de las variables `perimetro` y `area`:

```
// Obtiene el perímetro y el área de un rectángulo
// rectangulo.cpp

#include <iostream.h>

main() {
  // Declaración de las variables
  int base, altura;

  // Pedir los datos
  cout << "¿Cual es la base? ";
  cin >> base;
  cout << "¿Cual es la altura? ";
  cin >> altura;

   // Desplegar los resultados
  cout << "El perímetro es " << 2 * ( base + altura ) << endl
       << "El área es " << base * altura;

  return 0;
}
```

Solo usamos dos variables

Efectuamos las operaciones para obtener el área y el perímetro, dentro del `cout`

II.1.1 Ejercicios de estructura secuencial

Ejercicio 2.1.1.- Diseñar el diagrama de flujo y hacer el programa en C++ que obtenga la suma de dos números y la resta del primer número menos el segundo.

Solución:

1.- Determinar entradas y salidas (Figura II-5)

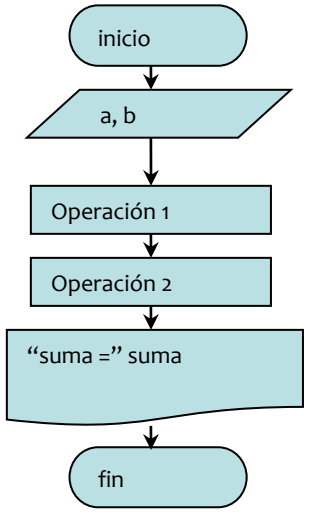


Figura II-5 *Obtención de suma y resta de dos números, solución parcial.*

2.- Especificar las operaciones uno y dos, el diagrama de flujo completo se muestra en la Figura II-6.

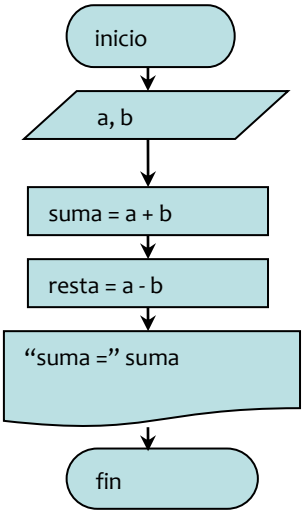


Figura II-6 *Obtención de suma y resta de dos números, solución completa.*

3.- Codificación: ¿Qué es lo que falta en el siguiente programa?

```
//Programa que suma y resta 2 números sumayresta.cpp
#include <iostream.h>

main() {
  int a, b, suma, resta;
  // Pedir datos
  cout << "introduzca el numero 1 ";
  cin >> a;
  cout << "introduzca el numero 2 ";
  cin >> b;
  // Hacer cálculos

  // Desplegar resultados

  return 0;
}
```

El programa completo que resuelve el ejercicio 1 es el siguiente.

```
//Programa que suma y resta 2 números sumayresta.cpp
#include <iostream.h>

main() {
  int a, b, suma, resta;

  // Pedir datos
  cout << "introduzca el numero 1 ";
  cin >> a;
  cout << "introduzca el numero 2 ";
  cin >> b;

  // Hacer cálculos
  suma = a + b;
  resta = a - b;
  // Desplegar resultados
  cout<<"La suma es " << suma << endl;
  cout << "la resta es " << resta;

  return 0;
}
```

Aquí también podríamos ahorrar variables efectuando los cálculos dentro del cout:

```
//Programa que suma y resta 2 números sumayresta.cpp
#include <iostream.h>

main() {
  int a, b;

  // Pedir datos
  cout << "introduzca el numero 1 ";
  cin >> a;
  cout << "introduzca el numero 2 ";
  cin >> b;

  // Desplegar resultados
  cout<<"La suma es " << a + b << endl;
  cout << "la resta es " << a - b;

  return 0;
}
```

Ejercicio 2.1.2.- Diseñar un algoritmo que obtenga la suma y el promedio de cinco números, codificarlo en un programa C++.

Solución:

1.- Determinar entradas y salidas (Figura II-7)

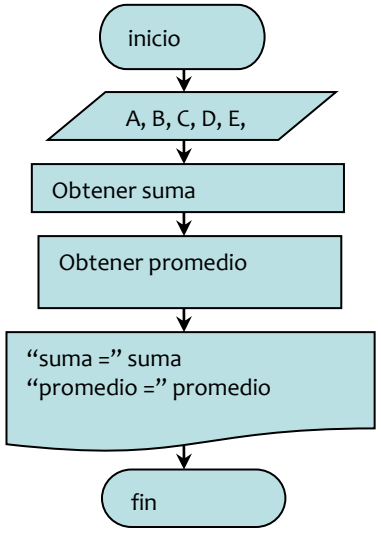


Figura II-7 *Entradas y salidas para obtener suma y promedio.*

2.- ¿Cómo se obtienen la suma y el promedio?, el algoritmo completo se muestra en la Figura II-8.

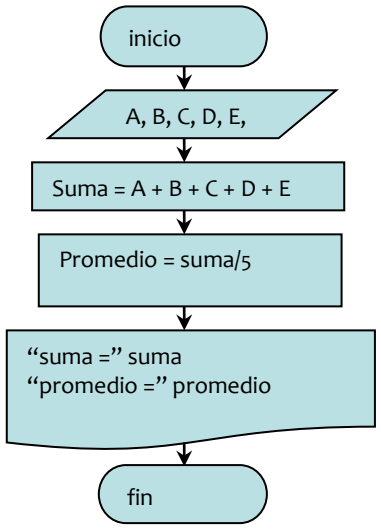


Figura II-8 *Entradas y salidas para obtener suma y promedio.*

3.- Codificación: ¿Qué es lo que falta en el siguiente programa en C++?

```
//obtiene el promedio de cinco números promedio.cpp
#include <iostream.h>

main() {
  float A, B, C, D, E, suma, promedio;
  // Pedir los cinco números

  // Obtener la suma y el promedio

  // Desplegar los resultados
  cout << "la suma es " << suma << endl;
  cout << "el promedio es " << promedio;

  return 0;
}
```

El programa completo se muestra a continuación:

```
//obtiene el promedio de cinco números promedio.cpp
#include <iostream.h>

main() {
  float A, B, C, D, E, suma, promedio;

  // Pedir los cinco números
  cout << "introduce el número 1 "; cin >> A;
  cout << "introduce el número 2 "; cin >> B;
  cout << "introduce el número 3 "; cin >> C;
  cout << "introduce el número 4 "; cin >> D;
  cout << "introduce el número 5 "; cin >> E;

  // Obtener la suma y el promedio
  suma = A+B+C+D+E;
  promedio = suma/5;

  // Desplegar los resultados
  cout << "la suma es " << suma << endl;
  cout << "el promedio es " << promedio;

  return 0;
}
```

Ejercicio 2.1.3.- Construir un algoritmo tal que, dado el costo de un artículo vendido (`precio`) y la cantidad de dinero entregado por un cliente (`pago`), calcule e imprima el `cambio` que debe entregarse al mismo. Codificarlo en C++. Considerar que el `pago` es siempre mayor o igual al `precio`.

Solución:

1.- Determinar entradas, operaciones y salida (Figura II-9)

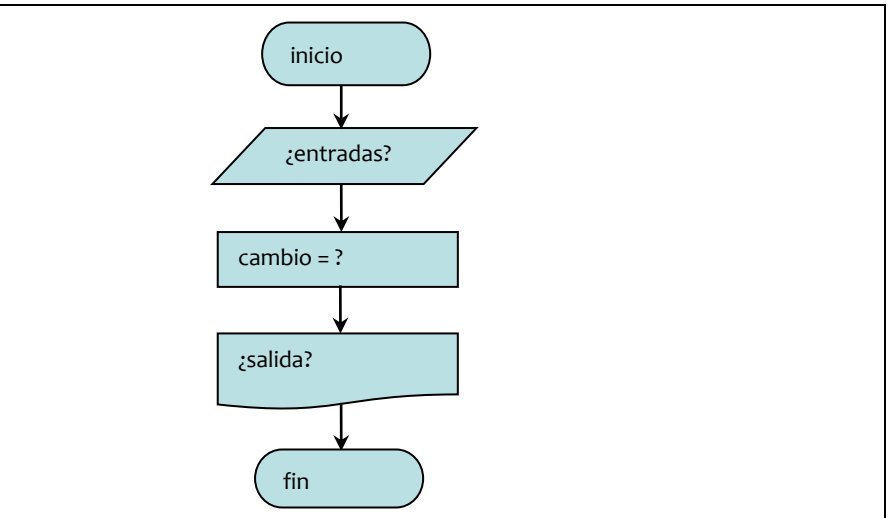


Figura II-9 *¿Cuáles son las entradas y salidas?.*

2.- ¿Cuáles son las variables de entrada? ¿Y la salida? ¿Cómo se obtiene el cambio? El algoritmo completo se muestra en la Figura II-10.

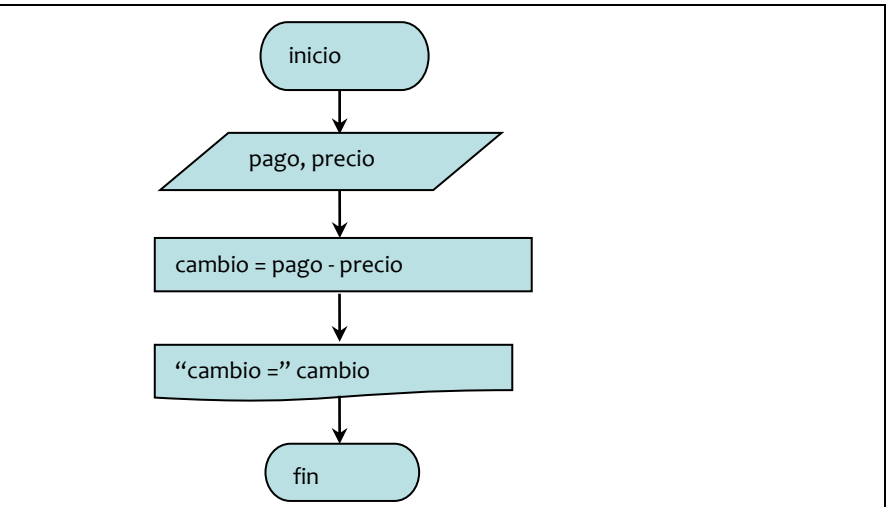


Figura II-10 *Algoritmo para obtener el cambio dado un precio y un pago.*

3.- Codificación: ¿Cómo se hace el programa en C++ para este algoritmo?

```
//Programa que calcula el cambio cambio1.cpp
#include <iostream.h>

float pago, precio, cambio;

main() {
  // Pedir datos
  cout << "Cual es el precio del producto? ";
  cin >> precio;
  cout << "De cuanto fue el pago? ";
  cin >> pago;

  cambio = pago - precio;

  cout << "El cambio es: " << cambio << endl;

  return 0;
}
```

Otra opción, con la que ahorramos la variable `cambio`, es la siguiente:

```
//Programa que calcula el cambio cambio1.cpp
#include <iostream.h>

float pago, precio;

main() {
  // Pedir datos
  cout << "Cual es el precio del producto? ";
  cin >> precio;
  cout << "De cuanto fue el pago? ";
  cin >> pago;

  cout << "El cambio es: "<< pago - precio << endl;

  return 0;
}
```

Ejercicio 2.1.4.- (Cilindro) Construir el diagrama de flujo y el programa en C++ para calcular el área de un cilindro la fórmula para calcular el área de un cilindro es:

```
areaBase = 2*pi*radio
areaCilindro = areaBase*altura
```

Solución:

1.- Determinar las entradas, las operaciones y la salida (Figura II-11).

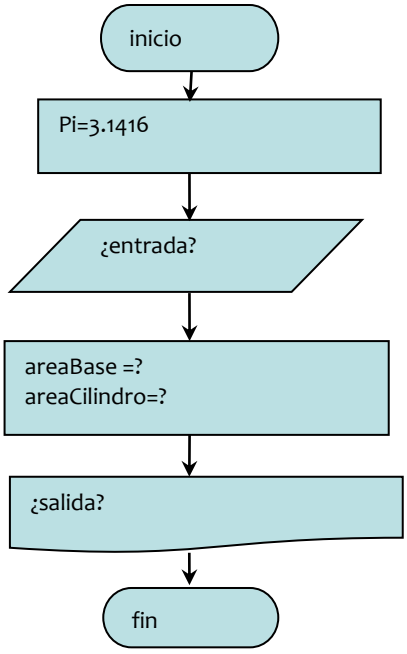


Figura II-11 *Determinar las entradas, las operaciones y la salida.*

2.- El algoritmo completo se muestra en la Figura II-12.

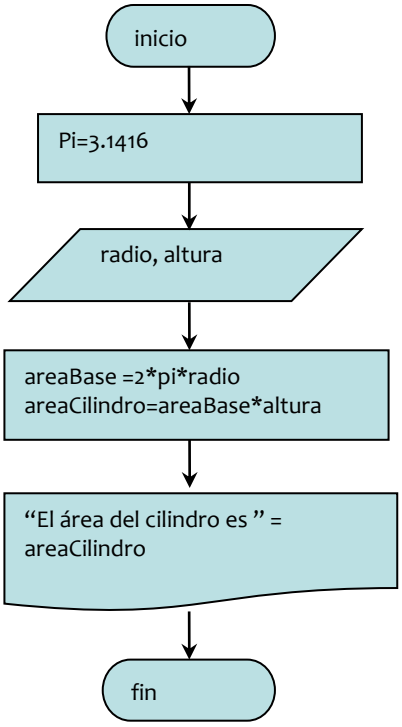


Figura II-12 *Algoritmo para obtener el área de un cilindro.*

3.- Codificación: ¿Qué le falta al siguiente programa en C++ para completar el algoritmo de la Figura II-12?

```
//Programa que calcula el área de un cilindro Cilindro.cpp
#include <iostream.h>
#include <math.h>

float radio, altura , areaBase, areaCilindro;
double M_PI;

main() {
  // Pedir datos

  // operaciones

  // desplegar resultaos

  system("PAUSE");
  return 0;
}
```

π Se encuentra declarada en la librería de math.h

A continuación presentamos el programa completo.

```
//Programa que calcula el área de un cilindro Cilindro.cpp
#include <iostream.h>
#include <math.h>

float radio, altura , areaBase, areaCilindro;
double M_PI;

main() {
  // Pedir datos
  cout <<"radio:\n ";
  cin >> radio;
  cout<< "altura: \n";
  cin >> altura;

  // operaciones
  areaBase = 2* M_PI *r;
  areaCilindro=areaBase*altura;

  // desplegar resultaos
  cout<<"El area del cilindro es:  "<<areaCilindro<<endl;

  system("PAUSE");
  return 0;
}
```

Ejercicio 2.1.5.- a)-Hacer el diagrama de flujo y el programa en C++ que convierta de grados a radianes. La fórmula para convertir de grados a radianes es:

```
Radianes = Grados*π/180
```

Solución:

1.- Determinar la entrada, la operación para convertir de grados a radianes y la salida (Figura II-13).

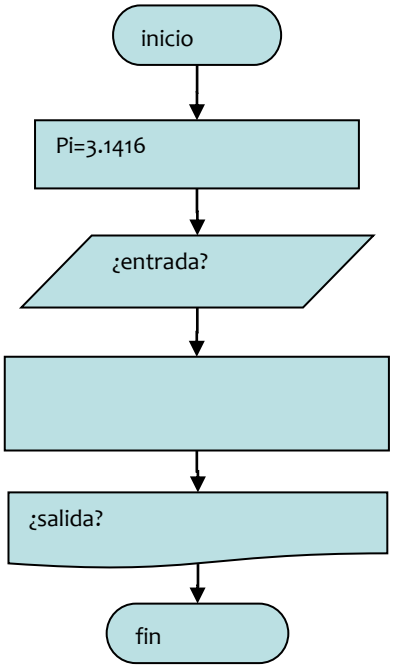


Figura II-13 *¿Cuál es la entrada y la salida?*

2.- El algoritmo completo se muestra en la Figura II-14.

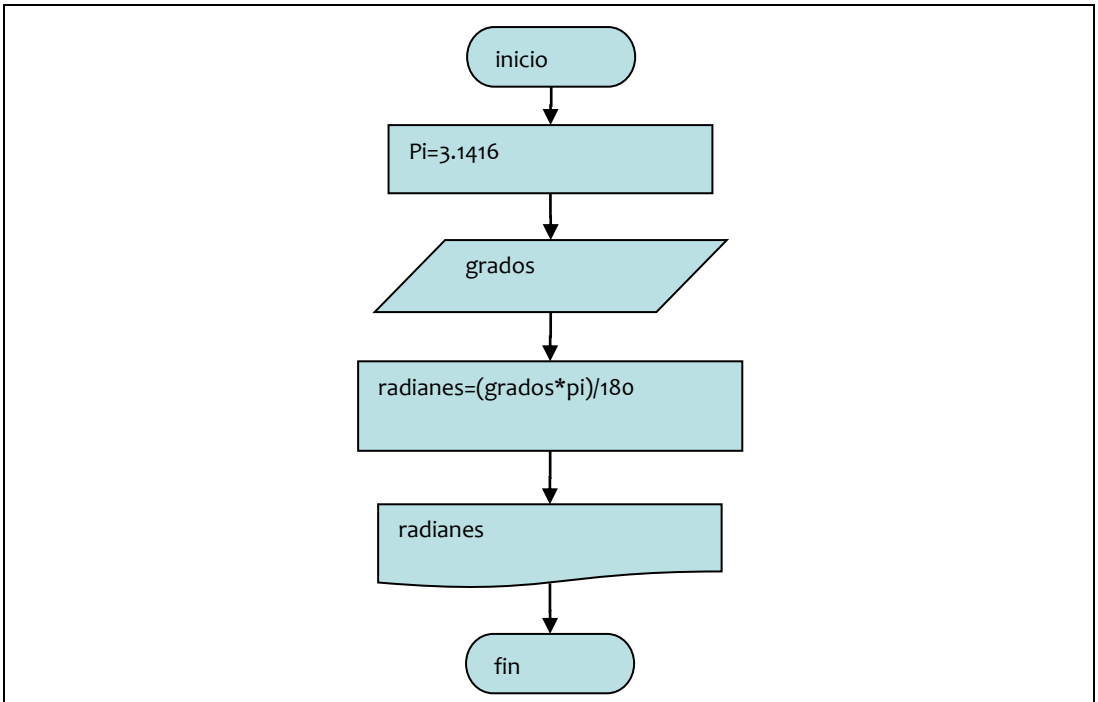


Figura II-14 *Algoritmo para convertir de grados a radianes*

3.- *Codificación*.- ¿Qué le falta al programa que se muestra a continuación para obtener el algoritmo del diagrama de la Figura II-14?

```
//Programa que convierte de grados a radianes
// gradosARadianes.cpp
#include <iostream.h>
#include<math.h>

float grados ,radianes;
double M_PI;

main() {
  // Pedir datos
  ...                    <---------  Pedir el dato en grados.

  radianes= ¿?;

  cout<< grados << " grados equivalen a: "
      << radianes << " radianes";

  system( "PAUSE" );
  return 0;
}
```

A continuación presentamos el programa completo.

```
//Programa que convierte de grados a radianes
// gradosARadianes.cpp
#include <iostream.h>
#include<math.h>

float grados ,radianes;
double M_PI;

main() {
  // Pedir datos
  cout<<"grados: ";
  cin>>grados;

  radianes=(grados*M_PI)/180;
  cout<< grados << " grados equivalen a: "
      << radianes << " radianes";
  system("PAUSE");
  return 0;
}
```

Ejercicio 2.1.6.- Hacer el diagrama de flujo y el programa en C++ que convierta de radianes a grados, la fórmula para convertir de radianes a grados es:

```
grados=(radianes*180)/pi;
```

Solución:

1.- Determinar la entrada, la operación para convertir de radianes a grados y la salida (Figura II-15).

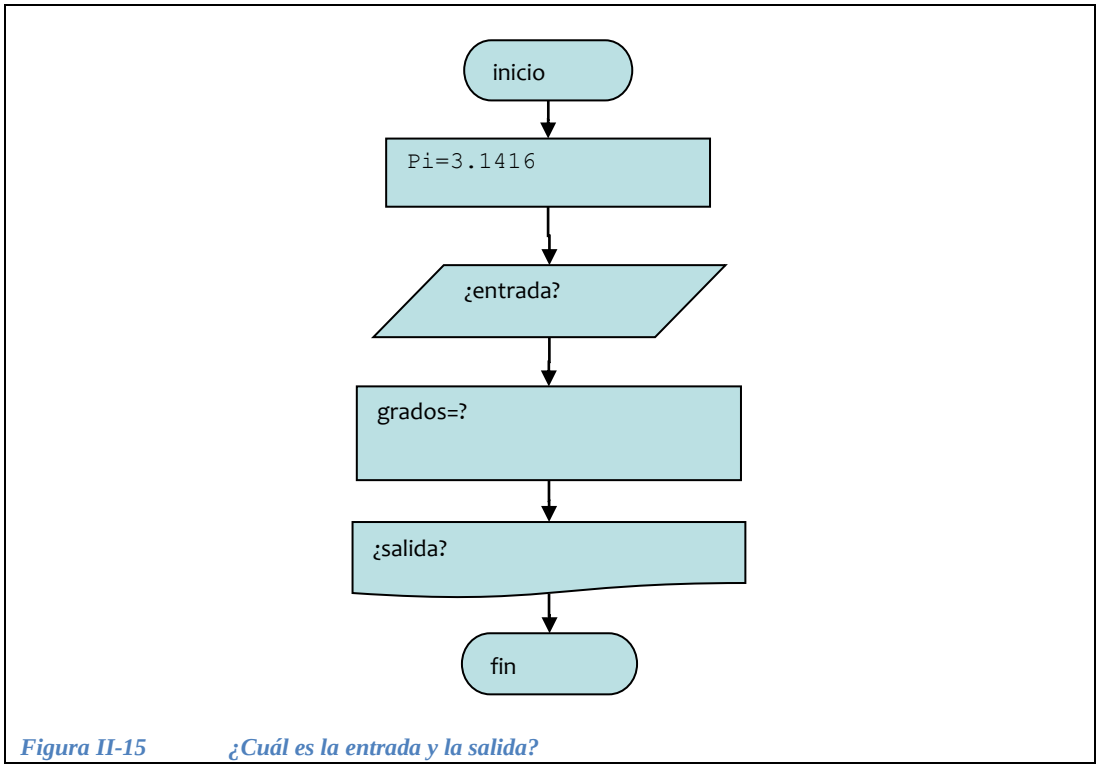


Figura II-15 *¿Cuál es la entrada y la salida?*

2.- ¿ El algoritmo completo se muestra en la Figura II-16.

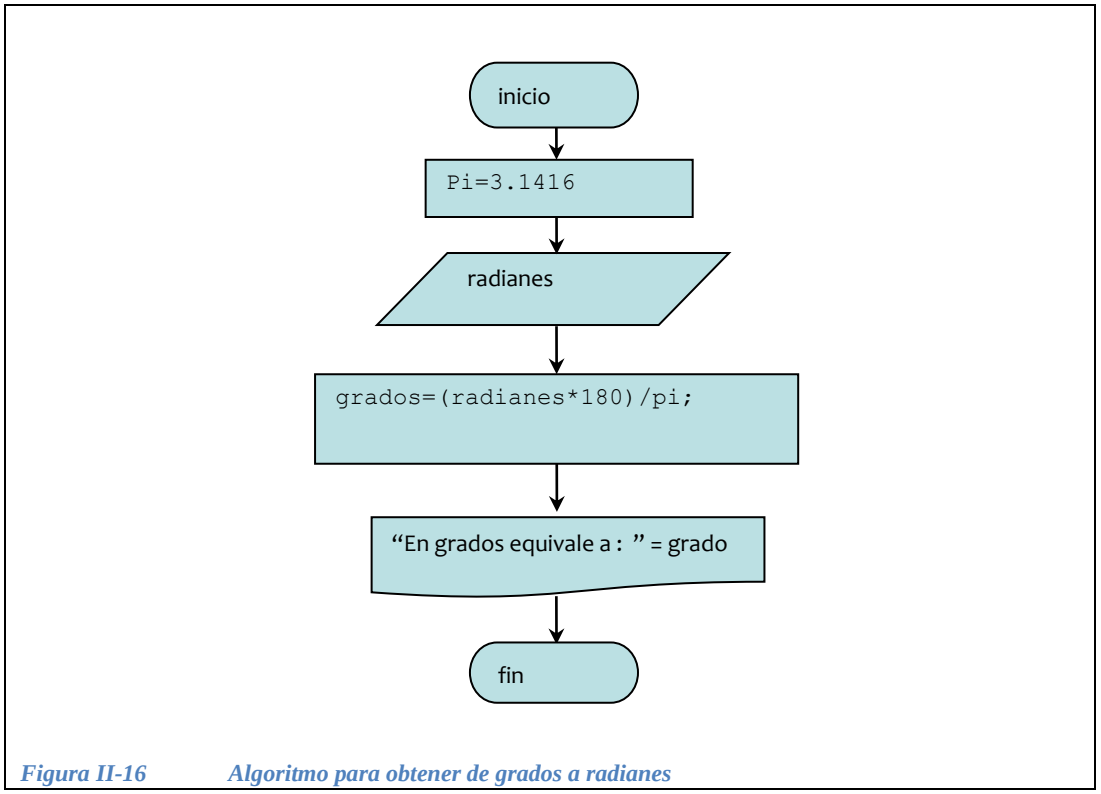


Figura II-16 *Algoritmo para obtener de grados a radianes*

3.- *Codificación*.- completar el siguiente programa en C++

```
//  Programa que convierte de radianes a grados
//  radianesAgrados.cpp
#include <iostream.h>
#include <math.h>

float grados ,radianes;
double M_PI;

main() {
  // Pedir dato (en radianes)
  ...
  grados = ?

  // Desplegar resultado
  ...

  system("PAUSE");
  return 0;
}
```

A continuación el programa completo:

```
//  Programa que convierte de radianes a grados
//  radianesAgrados.cpp
#include <iostream.h>
#include <math.h>

float grados ,radianes;
double M_PI;

main() {
  // Pedir dato (en radianes)
  cout<<"radianes: ";
  cin>>radianes ;

  grados=(radianes*180)/M_PI;

  // Desplegar resultado
  cout<< radianes << "radianes equivalen a: "
      << grados << " grados";

  system("PAUSE");
  return 0;
}
```

II.2 Estructura selectiva

La estructura selectiva. se usa cuando hay más de una posible *instrucción siguiente*. La elección de la siguiente instrucción depende de una *condición.*

II.2.1 Estructura selectiva sencilla

Para la estructura selectiva. sencilla: *Si* / *Entonces*, la *condición* es una operación cuyo resultado puede ser únicamente verdadero o falso.

La estructura selectiva. sencilla se representa en la Figura II-17:

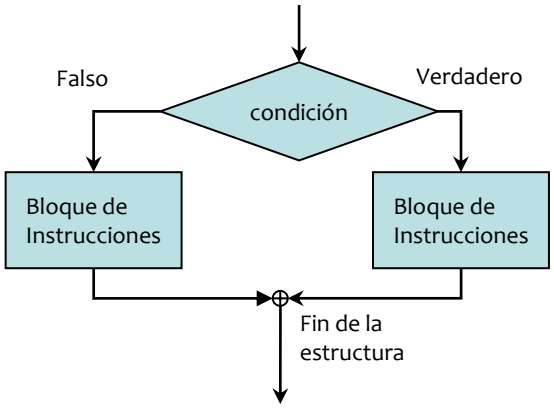


Figura II-17 *Estructura selectiva. sencilla, Si / Entonces.*

Los *operadores relacionales* permiten que los programas prueben como son unos valores con respecto a otros, la prueba consiste en verificar si un valor A es mayor, igual, o menor a cierto valor B. El resultado de esta comparación es "**verdadero**" o "**falso**".

En C++ `if/else` significa Si / Entonces. La instrucción `if` de C++ permite que los programas hagan una prueba y luego ejecuten instrucciones basándose en el resultado. La sintaxis en C++ de la instrucción `if` es la siguiente:

```
if( <condición> ){
  <bloque de instrucciones>;
};
```

El bloque de instrucciones se ejecuta siempre y cuando la condición sea verdadera.

La mayoría de las veces, los programas deben especificar un conjunto de instrucciones que se ejecuten cuando la condición sea cierta y otro conjunto o bloque de instrucciones para cuando la condición sea falsa. La sintaxis para especificar esto es la siguiente:

```
if( <condición> ){
  <bloque de instrucciones>;
} <----------------------  Nótese que aquí no lleva ";"
else {
  <bloque de instrucciones>;
};
```

La Figura II-18 es la respuesta a ciertas dudas recurrentes de algunos alumnos:

```
if( <expresión lógica> )
  <una sola instrucción;

if( <expresión lógica> ){
  <bloque de instrucciones>;
}

if( <expresión lógica> )
  <una sola instrucción;
else
  <una sola instrucción;

if( <expresión lógica> )
  <una sola instrucción;
else{
  <bloque de instrucciones>;
}

if( <expresión lógica> ){
  <bloque de instrucciones>;
}
else
  <una sola instrucción;

if( <expresión lógica> ){
  <bloque de instrucciones>;
}
else{
  <bloque de instrucciones>;
```

Figura II-18 Las diferentes formas del if/else en C y C++.

Ejemplo 2.2.1.- Diseñar un algoritmo para calcular el cambio de una venta pero consideremos ahora el caso en el que el pago puede ser inferior al precio. Cuando sucede este caso, hay que especificar el faltante en lugar del cambio. Para construir este algoritmo, es necesaria una estructura selectiva.

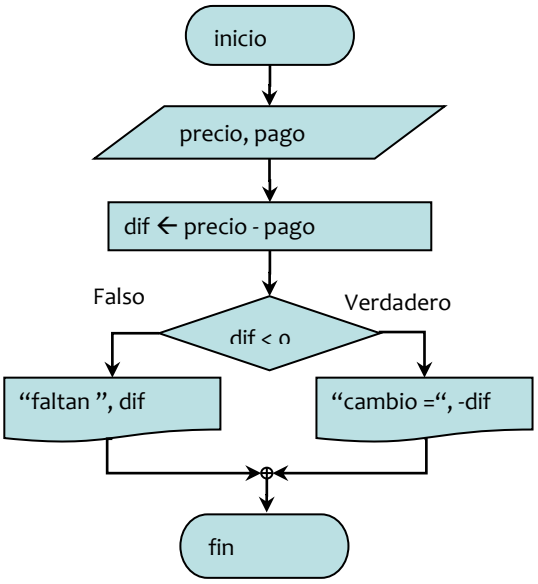


Figura II-19 *Ejemplo de estructura selectiva. sencilla.*

El ejemplo de la Figura II-19 se codifica en C++ de la siguiente forma:

```
// Programa para mostrar la estructura
// Selectiva. sencilla en C++
#include <iostream.h>

main() {
  float precio, pago, dif;

  cout << "precio:"; cin >> precio;
  cout << "pago:";   cin >> pago;

  dif = precio - pago;

  if(dif < 0)
    cout << "cambio =" << -dif << endl;

  else
    cout << "faltan " << dif << endl;

  return 0;
}
```

Aquí no lleva “ ; ”

Como es una sola instrucción lleva “ ; ”

Este “ ; ” es el fin de la instrucción `if`

A continuación se muestra el mismo programa pero usando un bloque de instrucciones, podemos observar que en este caso si son necesarios los corchetes.

```
// Ejemplo de estructura si/entonces con corchetes
#include <iostream.h>

main() {
  float precio, pago, dif;

  cout << "precio:"; cin >> precio;
  cout << "pago:";   cin >> pago;

  dif = precio - pago;

  if (dif < 0 ) {
    cout << "cambio =" ;
    cout << -dif << endl;
  }
  else {
    cout << "faltan"
    cout << dif << endl;
  };

  return 0;
}
```

Inicio de bloque de instrucciones

Aquí no lleva " ; "

Este es el fin de la instrucción `if`

Ejemplo 2.2.2.- Usando una estructura selectiva. sencilla, determinar si un alumno esta aprobado o reprobado en base a su calificación. Se aprueba con una calificación mayor o igual que seis.

La única entrada es la calificación, y las salidas serán dos mensajes, dependiendo de si el alumno esta aprobado o reprobado. ¿Cuál es la condición que falta en la Figura II-20?

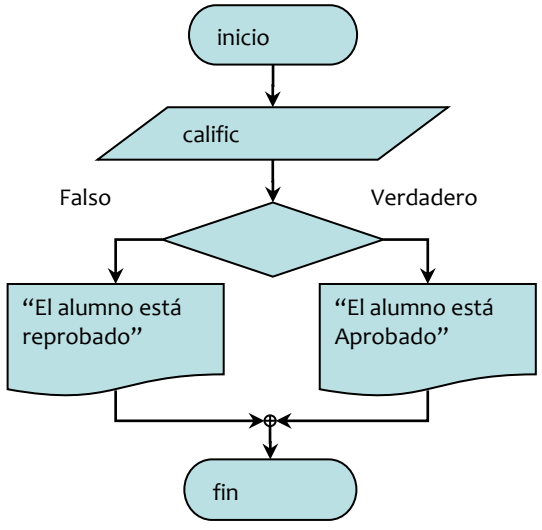


Figura II-20 Alumno aprobado o reprobado, solución parcial.

En la Figura II-21 se presenta el algoritmo completo.

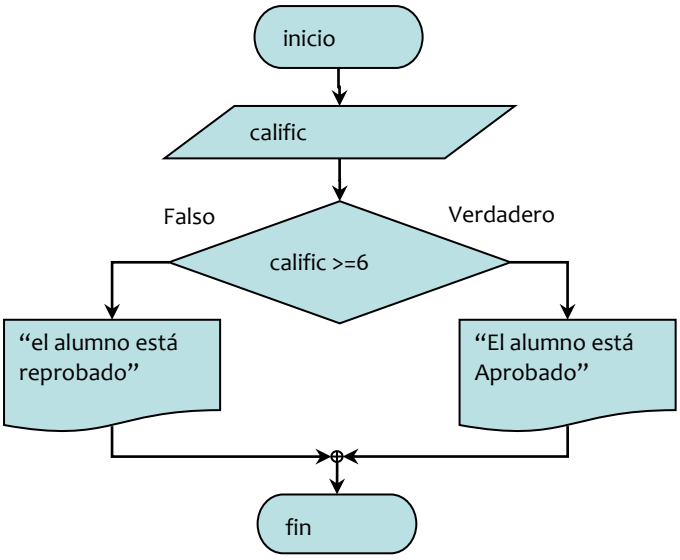


Figura II-21 *Alumno aprobado o reprobado, solución completa.*

El siguiente programa, es la codificación en C++ del algoritmo de la Figura II-21. Aquí podemos observar el uso de los corchetes porque hay más de una instrucción después del `if`, mientras que en el `else` se utilizó un solo `cout` y por lo tanto no fueron necesarios los corchetes. Se agregó el desplegado de la calificación en el mensaje de salida.

```
// Estructura selectiva. que da un mensaje
// mensaje.cpp
#include <iostream.h>

main() {
  int calific;

  cout << " cual es la calificación?" ";
  cin >> calific;

  if ( calific >= 6 ) {
    cout <<  "el alumno esta aprobado";
    cout << "su calificación es:" << calific << endl;
  }
  else
    cout << "el alumno esta reprobado, échele más ganas !"
         << "su calificación es:"<< calific << endl;

  return 0;
}
```

Inicio de bloque de instrucciones

Aquí no lleva " ; "

Este es el fin de la instrucción `if`

II.2.1.1 Ejercicios de estructura selectiva sencilla

Ejercicio 2.2.1.- Dado un número entero (`int`), determinar si es par o impar usando el criterio del módulo.

Solución:

1.- La entrada es un número entero al cual le llamaremos `num`, como se muestra en la Figura II-22, y la salida es uno de los dos mensajes señalados.

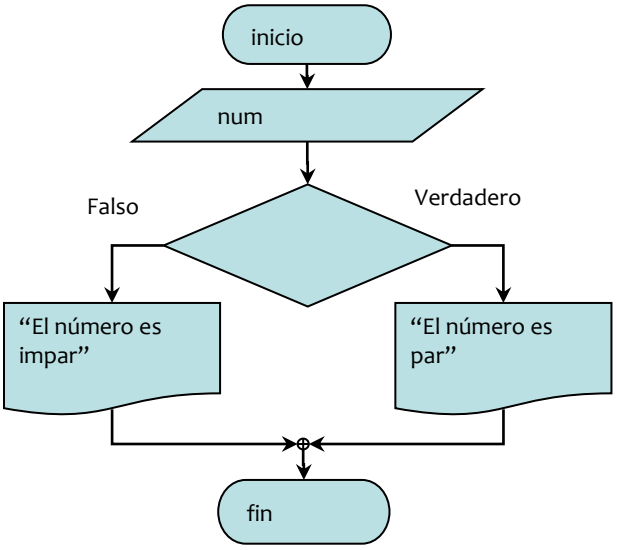


Figura II-22 ***Determinar si un número es par o impar, solución parcial.***

2.- Recordando los visto en el capítulo I, cuando tenemos `x = A % y`, se lee: *x* es *A módulo y,* donde *x* es el residuo que se produce al hacer una división de *A / y*. El módulo 2, es muy útil para saber si un número es par o impar, ya que cuando dividimos un número **par** entre dos el *residuo siempre es cero*, y si dividimos un número **impar** entre dos *el residuo siempre es uno*. Teniendo en cuenta lo anterior ¿cómo se expresaría la condición para saber si un número entero es par o impar en la *figura4.2.5.a*? El algoritmo completo se muestra en la Figura II-23.

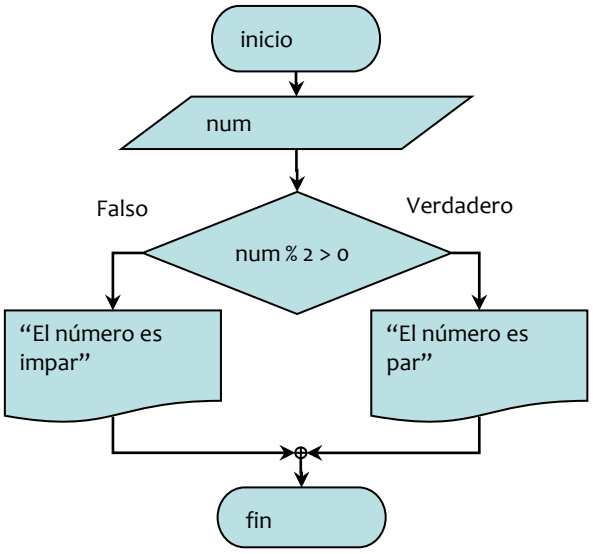


Figura II-23 *Criterio del módulo para saber si un número es par o impar.*

3.- Declarar lo necesario y pedir el dato. ¿Cómo se codifica la condición en el siguiente programa?

```
// Programa que determina si un número entero es par o impar
// par1.cpp
#include <iostream.h>

main() {
  // Declaración
  ...

  // Pedir el número
  ...

  if( ... )
    cout << "El número: " << num << "  es par";

  else
    cout << "El número: " << num << "  es impar";

  return 0;
}
```

A continuación, el programa completo.

```
//Programa que determina si un número entero es par o impar
par1.cpp
#include <iostream.h>

main() {
  int num;

  // Pedir el número
  cout << "Cual es el numero? ";
  cin >> num;

  if( num%2 == 0 )
    cout << "El número: " << num << "  es par";

  else
    cout << "El número: " << num << "  es impar"

  return 0;
}
```

Nótese que para comparaciones se usa doble =

Ejercicio 2.2.2.- Dado un número entero (int), determinar si es par o impar usando el criterio de la potencia.

Solución:

1.- Las entradas y salidas son las mismas que las del ejercicio anterior, lo único que cambia es la condición.

2.- Operaciones. El -1, elevado a un número **par** siempre es *positivo*. Si se eleva -1 a un número **impar** el resultado siempre es *negativo*, ¿cómo se expresaría esta condición en la Figura II-24? El algoritmo completo se muestra en la Figura II-25.

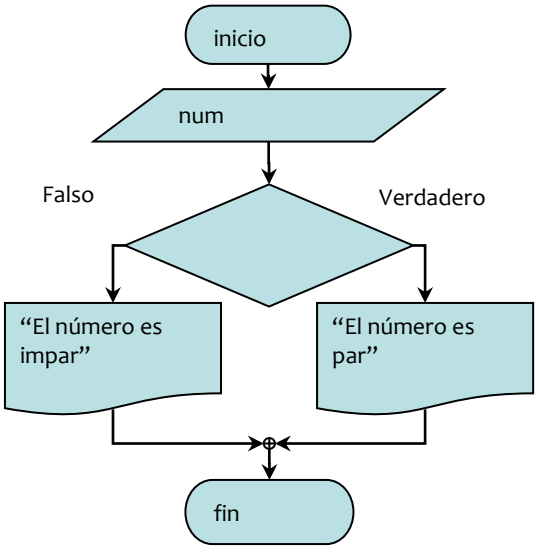


Figura II-24 *Algoritmo para saber si un número es par o impar.*

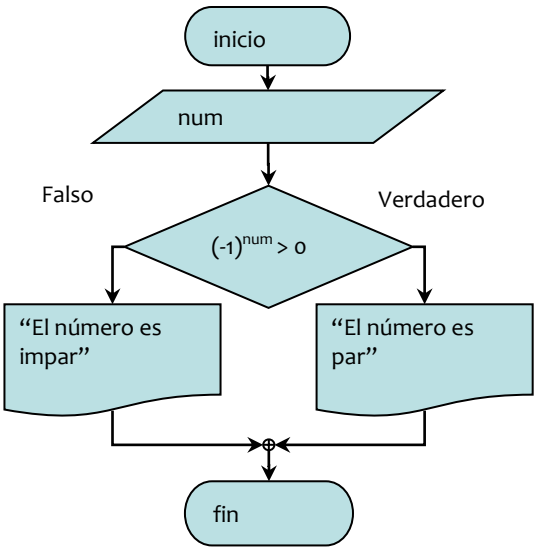


Figura II-25 *Criterio de la potencia para saber si un número es par o impar.*

3.- Para codificar el algoritmo anterior, usar la función `pow(base,exponente)` vista en el capítulo III. Recordar que es necesario incluir la librería `<math.h>`.

```
//Programa que determina si un número entero es par o impar
// par2.cpp
#include <iostream.h>
#include <math.h>

main() {
  int num;

  cout << "Cual es el numero? ";
  cin >> num;
  if( pow( -1, num )  > 0 )
    cout << "El número: ” << num << ”  es par”;

  else
    cout << "El número: ” << num << ”  es impar”;

  return 0;
}
```

Ejercicio 2.2.3.- Dados dos números reales (`float`) diferentes entre sí, determinar cuál de ellos es el mayor.

Solución:

1.- Las entradas son dos números reales a los que llamaremos `num1` y `num2`, y la salida será el número mayor, como se muestra en la Figura II-26.

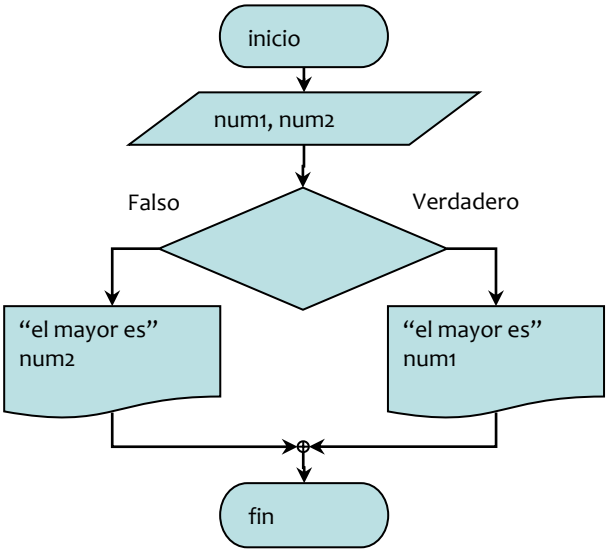


Figura II-26 *Determinar cual número es mayor, solución parcial.*

2.- ¿Cuál es la condición que debe ir en la Figura II-26?

El algoritmo completo se muestra en la Figura II-27.

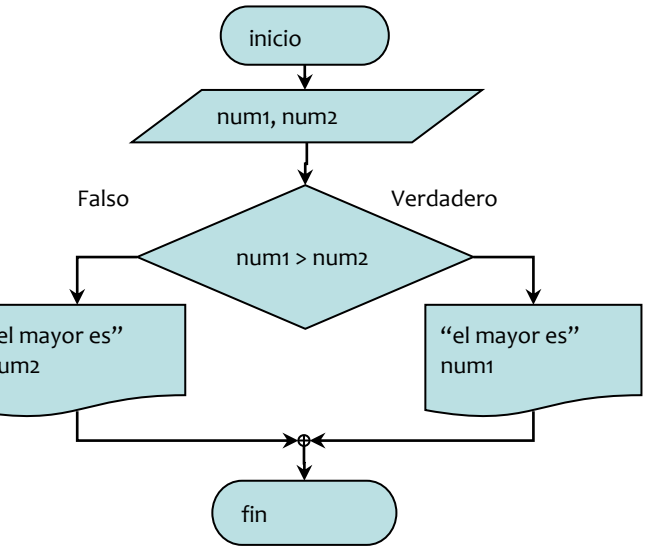


Figura II-27 *Determinar cual número es mayor, solución completa.*

3.- Codificación: ¿Qué es lo que falta en el siguiente programa?

```
//Programa que indica cual es el mayor de 2 números
// Diferentes entre sí mayor1.cpp
#include <iostream.h>

main() {
  // Declarar los números
  ...

  // Pedir los dos números
  ...

  if(...)
    cout << "el mayor es " << num1;
  else
    cout << "el mayor es " << num2;

  return 0;
}
```

A continuación el programa completo en C++.

```
//Programa que indica cual es el mayor de 2 números
// Diferentes entre sí mayor1.cpp
#include <iostream.h>

main() {
  float num1, num2;

  cout << "introduzca 2 números diferentes " << endl;
  cin >> num1;
  cin >> num2;

  if( num1 > num2 )
    cout << "el mayor es " << num1;
  else
    cout << "el mayor es " << num2;

  return 0;
}
```

Ejercicio 2.2.4.- Hacer el diagrama de flujo y el programa para obtener las raíces de una ecuación de segundo grado de la forma $ax^2 + bx + c$ si $a \neq 0$. La fórmula para resolver esta ecuación es la siguiente $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ existen tres casos posibles:

$$disciminante = b^2 - 4ac \begin{cases} > 0 & \text{Dos soluciones distintas} \\ = 0 & \text{Una solución doble } \dfrac{-b}{2a} \\ < 0 & \text{No tiene solución real} \end{cases}$$

Ejemplo: $2x^2 + 4x - 6 = 0$ discriminante mayor de cero, dos soluciones distintas

$$\begin{array}{l} a = 2 \\ b = 4 \\ c = -6 \end{array} \left| \; x = \frac{-4 \pm \sqrt{4^2 - 4 \cdot 2 \cdot (-6)}}{2 \cdot 2} \Rightarrow x = \frac{-4 \pm \sqrt{64}}{4} \Rightarrow x = \frac{-4 \pm 8}{4} \begin{cases} x_1 = \frac{-4+8}{4} \rightarrow x_1 = 1 \\ x_2 = \frac{-4-8}{4} \rightarrow x_2 = -3 \end{cases} \right.$$

Ejemplo: $x^2 + 2x + 1 = 0$ discriminante igual a cero, una solución doble

$$\begin{array}{l} a = 1 \\ b = 2 \\ c = 1 \end{array} \left| \; x = \frac{-2 \pm \sqrt{2^2 - 4 \cdot 1 \cdot 1}}{2 \cdot 1} \Rightarrow x = \frac{-2 \pm \sqrt{0}}{2} \Rightarrow x = \frac{-2}{2} \Rightarrow x = -1 \right.$$

Ejemplo: $2x^2 + 2x + 1 = 0$ discriminante menor que cero, no tiene solución real

$$\begin{array}{l} a = 2 \\ b = 2 \\ c = 1 \end{array} \left| \; x = \frac{-2 \pm \sqrt{2^2 - 4 \cdot 2 \cdot 1}}{2 \cdot 1} \Rightarrow x = \frac{-2 \pm \sqrt{-4}}{2} \Rightarrow \text{no tiene solución} \right.$$

Las *entradas* son los coeficientes a, b, y c y las *salidas* son las raíces x_1 y x_2 . ¿Cómo completarías el siguiente diagrama de flujo? (ver la Figura II-28) ¿Cómo indicar cuando las raíces son reales y cuando son complejas?

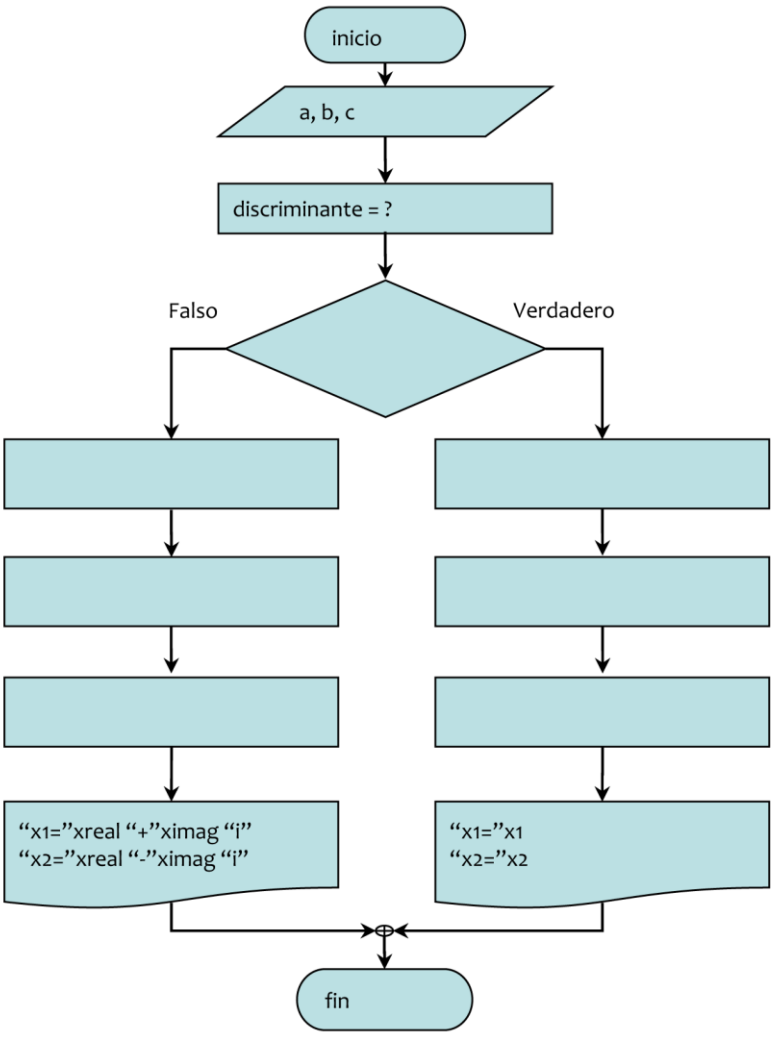


Figura II-28 *Diagrama de flujo para calcular las raices de un polinomio de grado 2.*

Solución:

1.- Las entradas son 3 números reales a los que llamaremos `a` `b` y `c`, y la salida depende del determinante. Si el determinante es mayor o igual a cero las raíces son reales y si es menor a cero, entonces son complejas. Ver Figura II-29.

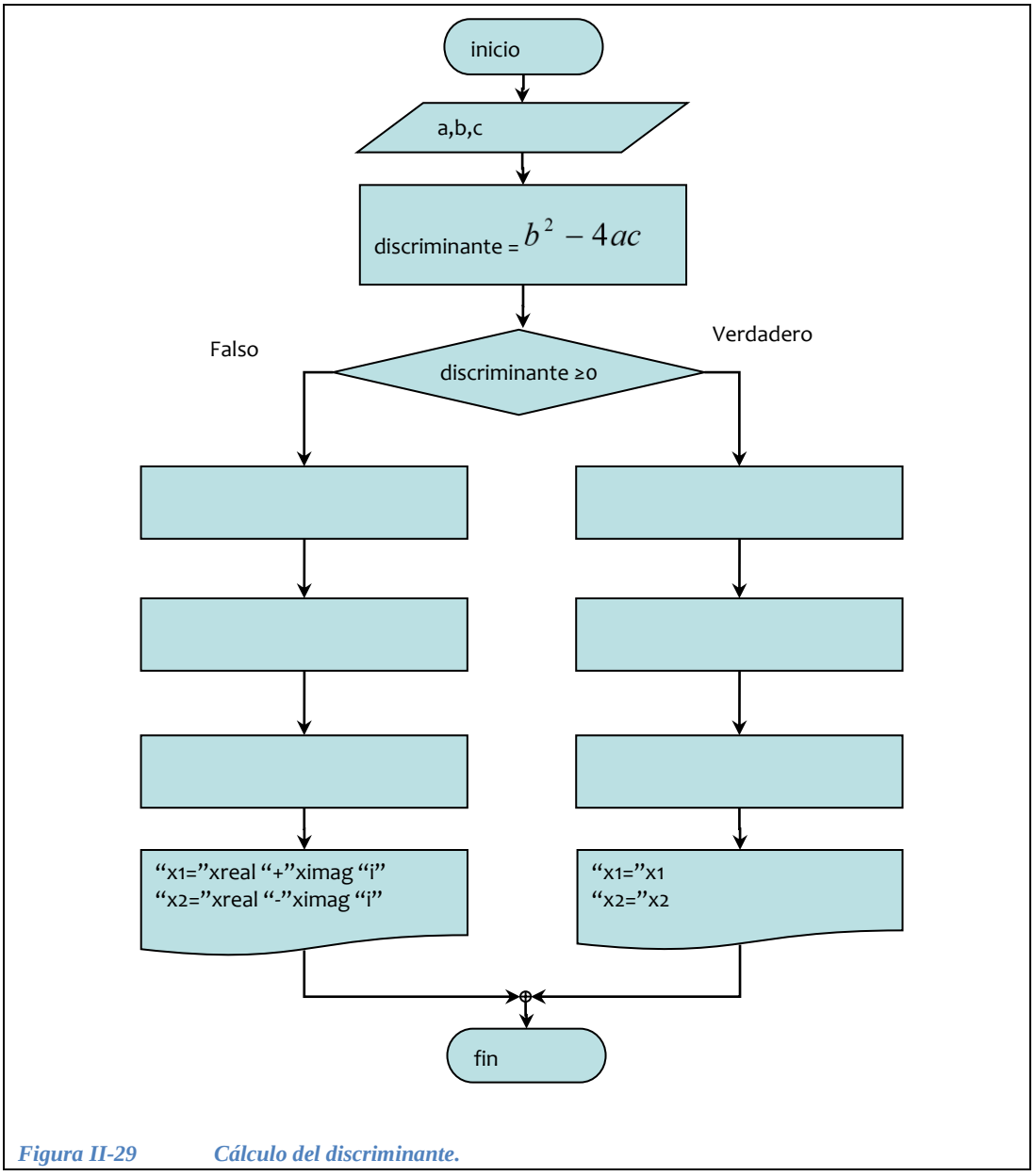


Figura II-29 *Cálculo del discriminante.*

2.- ¿Cómo se calculan las raíces reales? Ver Figura II-30.

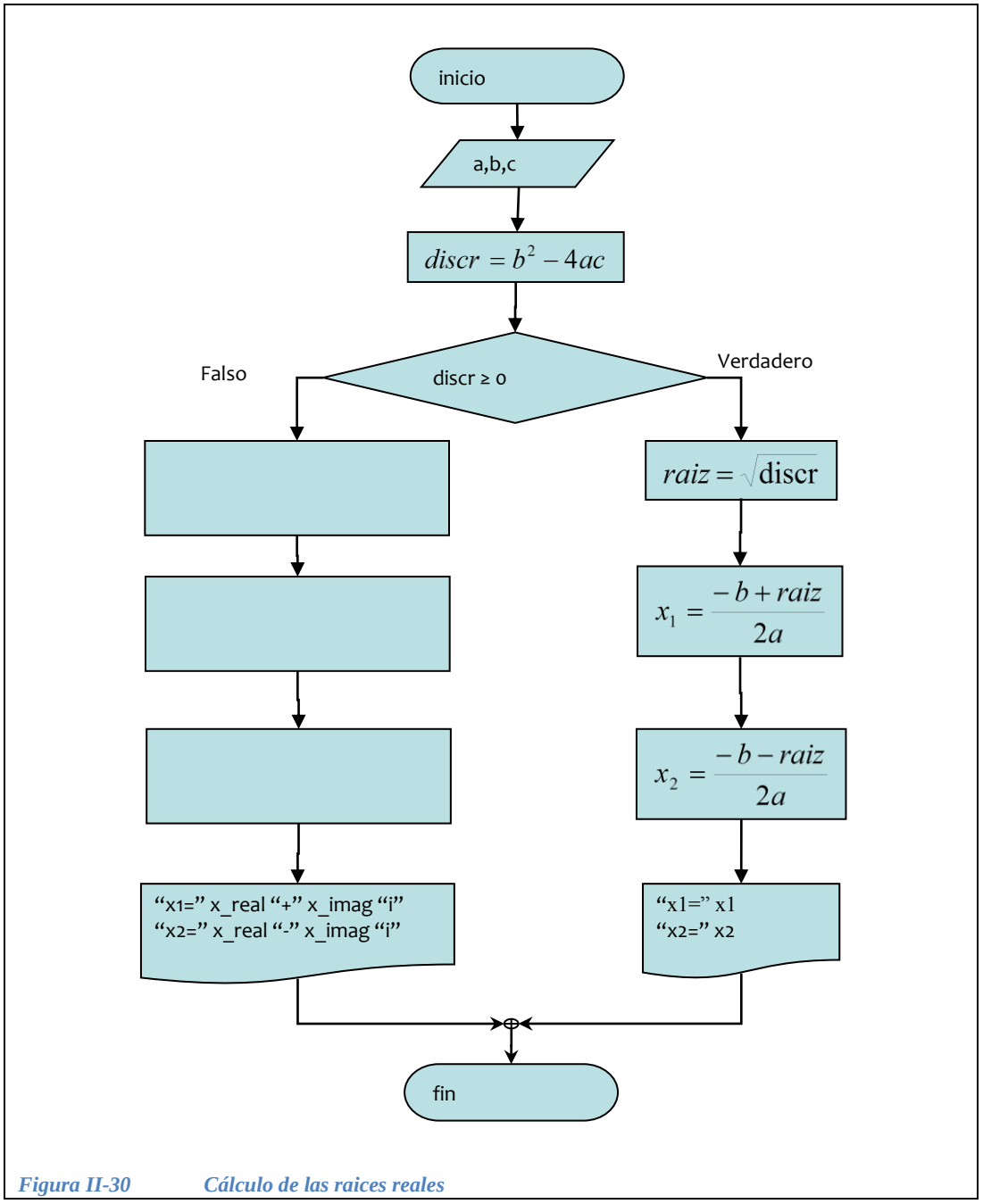


Figura II-30 *Cálculo de las raices reales*

3.- ¿Cómo se calculan las raíces complejas? Ver Figura II-31.

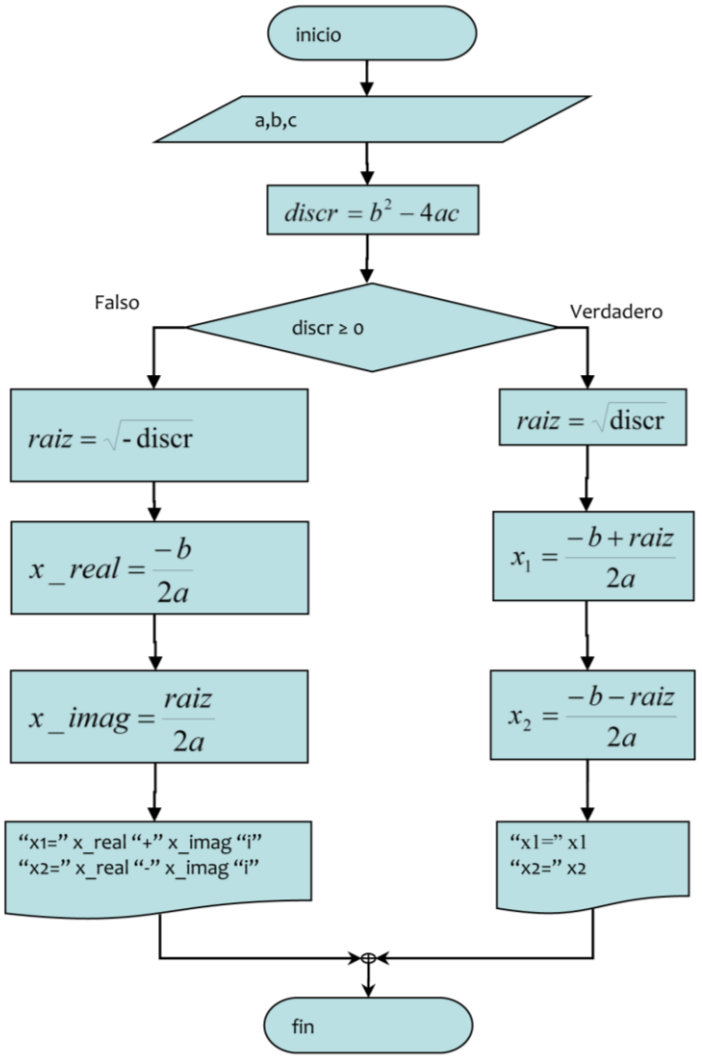


Figura II-31 *Obtención las raíces de una ecuación de segundo grado.*

5.- *Codificación*: A continuación presentamos una primera aproximación del código que resuelve una ecuación de segundo grado dados como entradas los coeficientes `a`,`b` y `c` ¿Qué le hace falta?

```
// Programa que obtiene las raíces de una
// ecuación de segundo grado.
// ecuacion2.ccp
#include<math.h>
#include<iostream.h>

main() {
  double a,b,c,x1,x2,discriminante,raiz,xreal,ximag;
  cout<<"\n Cual es el valor de a?";
  cin>>a;
  cout<<"\n Cual es el valor de b?";
  cin>>b;
  cout<<"\n Cual es el valor de c?";
  cin>>c;

  discriminante =?;
  if(condición) {

    // Calcular las raíces reales

   cout<<"\nLas raíces reales son= ";
   cout<<"\n x1="<<x1;
   cout<<"\n x2="<<x2<<"\n";

  }
  else {
    // Calcular las raíces complejas

    cout<<"\n x1="<<xreal<<" + "<<ximag<<" i ";
    cout<<"\n x2="<<xreal<<" - "<<ximag<<" i "<<endl;
  }

  system("PAUSE");
  return 0;
}
```

¿Cuál es la condición que determina si las raíces son reales o complejas?

Aquí hacen falta las operaciones para obtener las raíces reales

Aquí hacen falta las operaciones para obtener las raíces complejas

6.- Presentamos como se obtiene el discriminante y la condición. ¿Cómo calcularías las raíces reales y las complejas?

```
// Programa que obtiene las raíces de una
// ecuación de segundo grado.
// ecuacion2.ccp
#include<math.h>
#include<iostream.h>

main() {
  double a,b,c,x1,x2,discriminante,raiz,xreal,ximag;
  cout<<"\n Cual es el valor de a?";
  cin>>a;
  cout<<"\n Cual es el valor de b?";
  cin>>b;
  cout<<"\n Cual es el valor de c?";
  cin>>c;
  discriminante = pow(b,2)-(4*a*c);
  if(discriminante >=0) {
   // Calcular las raíces reales

                    <------  Aquí hacen falta las operaciones para
                             obtener las raíces reales

    cout<<"\nLas raices reales son= ";
    cout<<"\n x1="<<x1;
    cout<<"\n x2="<<x2<<"\n";
  }
  else {
    // Calcular las raíces complejas
                              <------  Aquí hacen falta las operaciones para
                                       obtener las raíces complejas

    cout<<"\nLas raíces complejas son= ";
    cout<<"\n x1="<<xreal<<" + "<<ximag<<" i ";
    cout<<"\n x2="<<xreal<<" - "<<ximag<<" i "<<endl;
  }

  system("PAUSE");
  return 0;
}
```

7.- A continuación el programa completo en C++.

```
// Programa que obtiene las raíces de una
// ecuación de segundo grado.
// ecuacion2.ccp
#include<math.h>
#include<iostream.h>

main() {
  double a,b,c,x1,x2,discriminante,raiz,xreal,ximag;
  cout<<"\n Cual es el valor de a?";
  cin>>a;
  cout<<"\n Cual es el valor de b?";
  cin>>b;
  cout<<"\n Cual es el valor de c?";
  cin>>c;
  discriminante = pow(b,2)-(4*a*c);
  if(discriminante >=0) {
    // Calcular raíces reales
    raiz=sqrt(discriminante);
    x1=(-b+raiz )/(2*a);
    x2=(-b-raiz )/(2*a);
    cout<<"\nLas raices reales son= ";
    cout<<"\n x1="<<x1;
    cout<<"\n x2="<<x2<<"\n";
  }
  else {
    // Calcular raíces complejas
    raiz=sqrt(-discriminante);
    xreal=(-b)/(2*a);
    ximag=raiz/(2*a);
    cout<<"\nLas raíces complejas son= ";
    cout<<"\n x1="<<xreal<<" + "<<ximag<<" i ";
    cout<<"\n x2="<<xreal<<" - "<<ximag<<" i "<<endl;
  }

  system("PAUSE");
  return 0;
}
```

II.2.2 Estructuras selectivas anidadas

En numerosos casos, para solucionar un problema, luego de tomar una decisión y marcar el camino correspondiente a seguir, es necesario tomar otra decisión anidada dentro de la primera.

Puede haber varios niveles de anidamiento dependiendo del problema que se está resolviendo.

Ejemplo 2.2.3.- Se pide la edad de un muchacho y de una muchacha, a estos datos les llamaremos `chavo` y `chava`. Hacer un algoritmo que de las siguientes recomendaciones sobre su hora de regreso del antro:

- Si los dos son mayores de edad regresan a la hora que quieran.
- Si el chavo es mayor de edad y la chava no, regresan a las dos de la mañana.

- Si la chava es mayor de edad y el chavo no, regresan a las doce de la noche.
- Si los dos son menores de edad no van.

Las entradas son las edades `chavo` y `chava`, y las salidas son cuatro mensajes diferentes, para cada uno de los casos del planteamiento del problema. En la Figura II-32 observamos la solución parcial del algoritmo. ¿Cómo lo completarías?

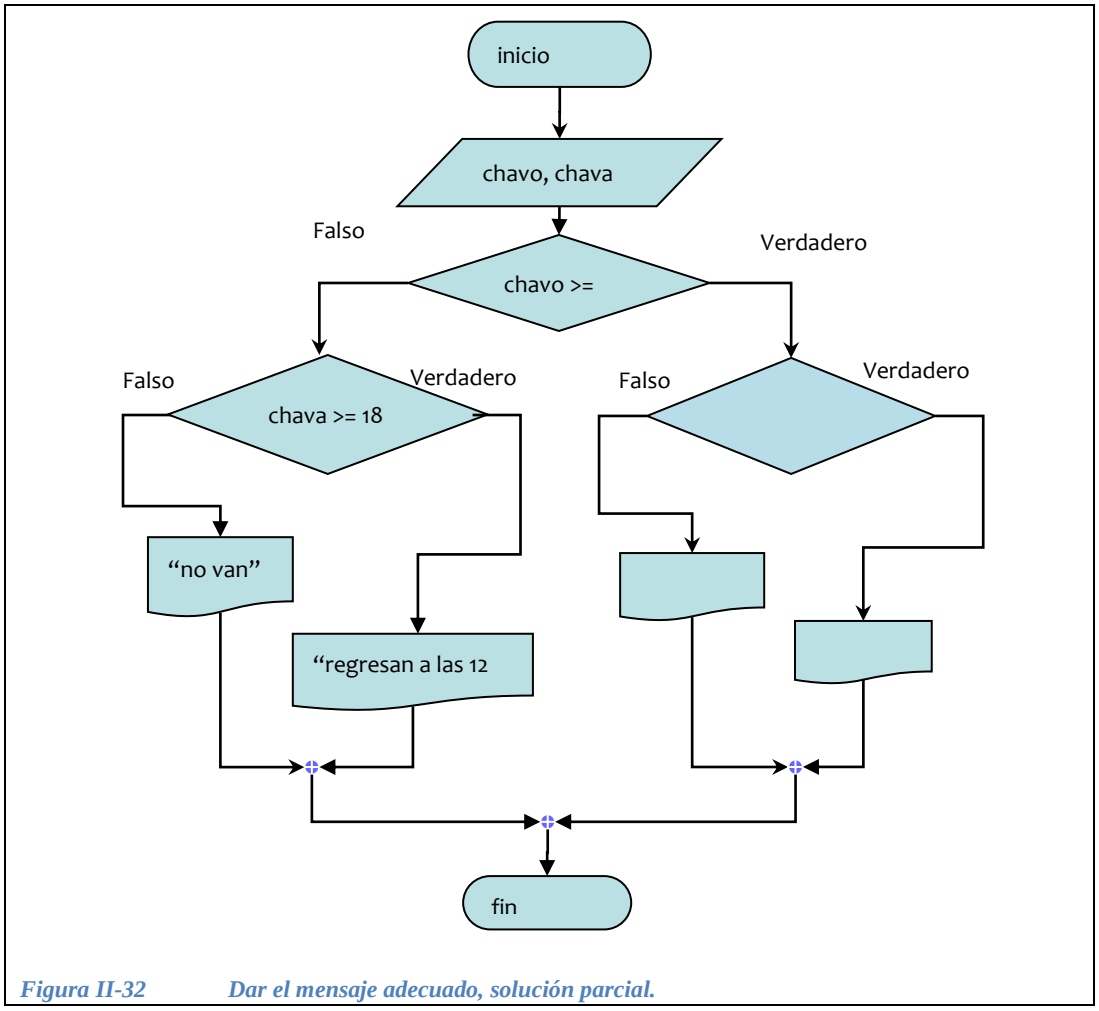


Figura II-32 *Dar el mensaje adecuado, solución parcial.*

Si la edad del chavo es menor a 18, entonces "no van" o "regresan a las 12", dependiendo si la chava es mayor de edad o no. Hace falta contemplar el caso en el que el chavo es mayor de edad, entonces, dependiendo de la edad de la chava "regresan a las 2 a.m." o "regresan cuando quieran". La solución completa de este algoritmo esta en la Figura II-33.

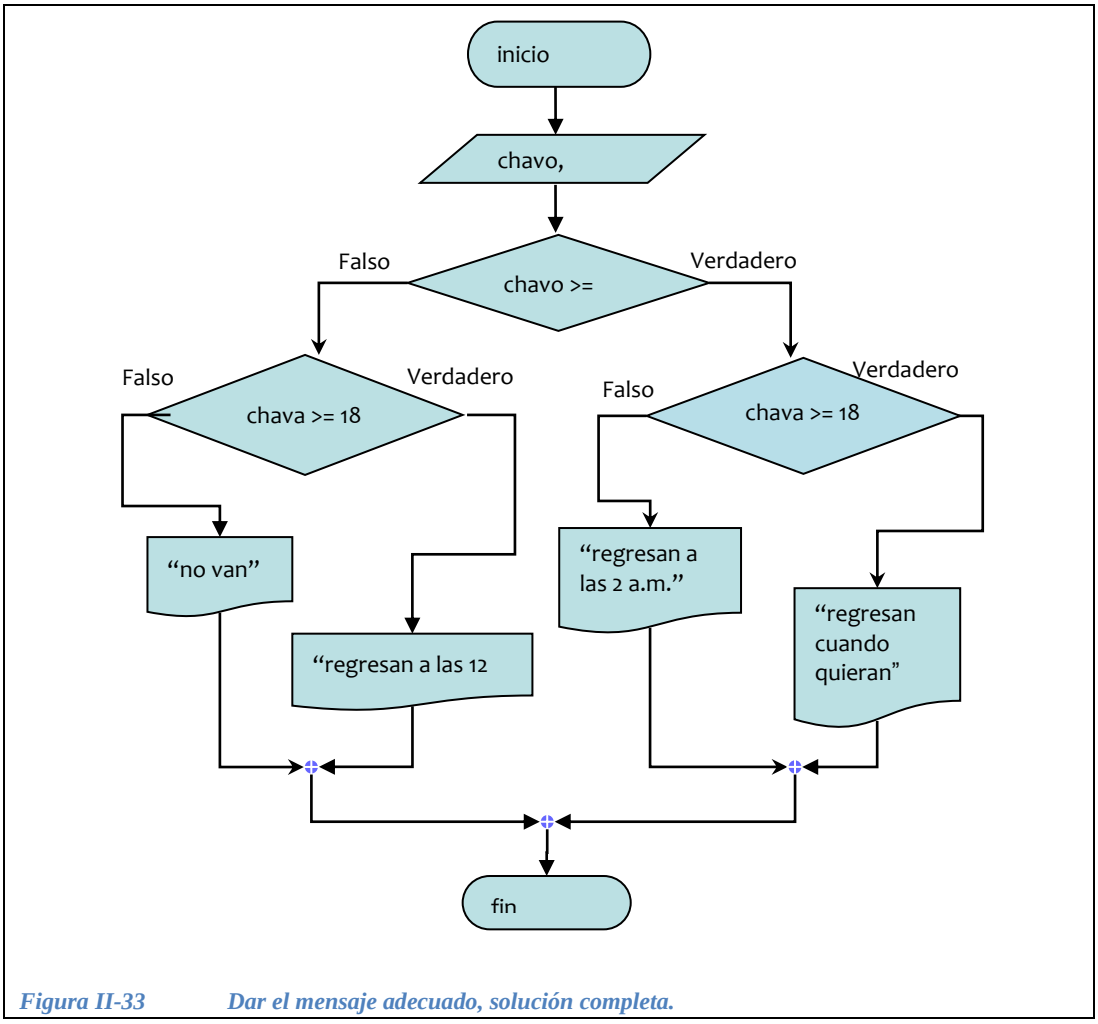
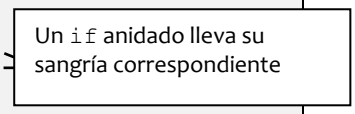


Figura II-33 *Dar el mensaje adecuado, solución completa.*

A continuación se muestra el código en C++ para resolver el problema del ejemplo 1.

```
// Da una orden en función de la edad de los muchachos antro.cpp
#include <iostream.h>

main()
{
  int chavo, chava;
  // pedir lo datos
  cout << "Que edad tiene el chavo?";  cin >> chavo;
  cout << "Que edad tiene la chava?";  cin >> chava;

  if( chavo >= 18 ) {
    if( chava >= 18 )
      cout << " regresen cuando quieran ";
    else
      cout << " regresen a las dos de la mañana";
  }
  else {
    if( chava >= 18 )
      cout << " regresen a las 12 a.m. ";
    else
      cout << " no van ";
  };

  return 0;
}
```



II.2.2.1 Ejercicios de estructuras selectivas anidadas.

Ejercicio 2.2.5.- Dados dos números reales (`float`) determinar cuál de ellos es el mayor tomando en cuenta que los números pueden ser iguales.

Solución:

1.- Las entradas son dos números reales a los que llamaremos `num1` y `num2`, y la salida será el número mayor.

2.- Es necesario una estructura selectiva. anidada, para contemplar el caso en el que los dos números son iguales. En la Figura II-34 se proporciona la solución parcial. Observar que en el caso en el que los números no sean iguales entre sí, el algoritmo es el mismo que el de la Figura II-27.

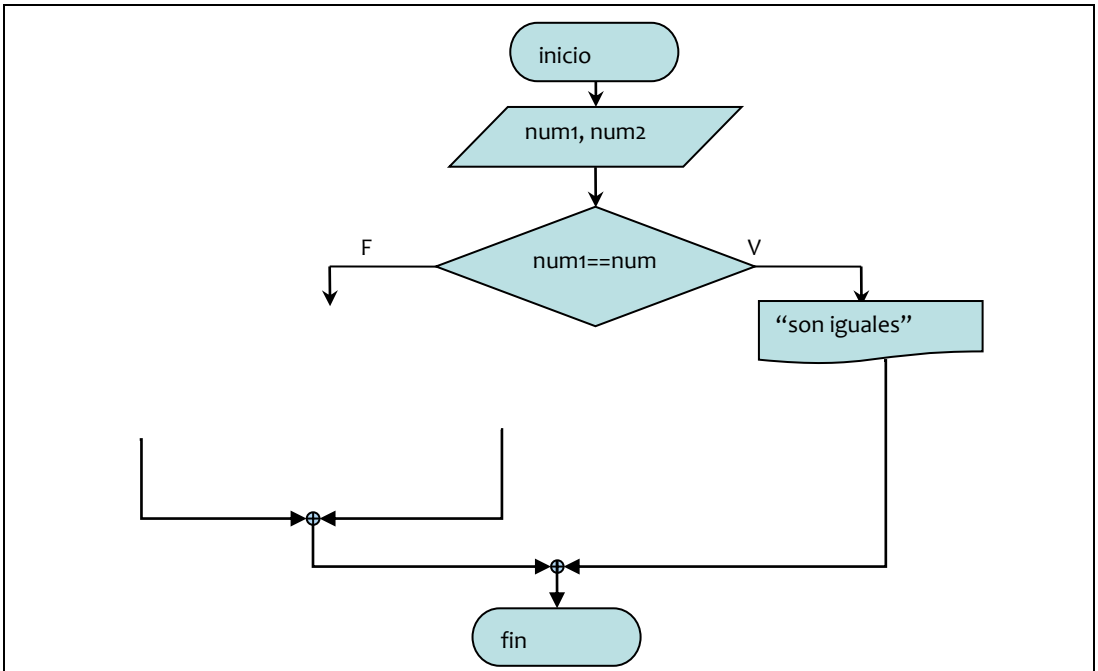


Figura II-34 *Determinar si hay un numero mayor o si son iguales, sol. Parcial.*

El algoritmo de la figura Figura II-27 queda anidado dentro del algoritmo de la Figura II-35.

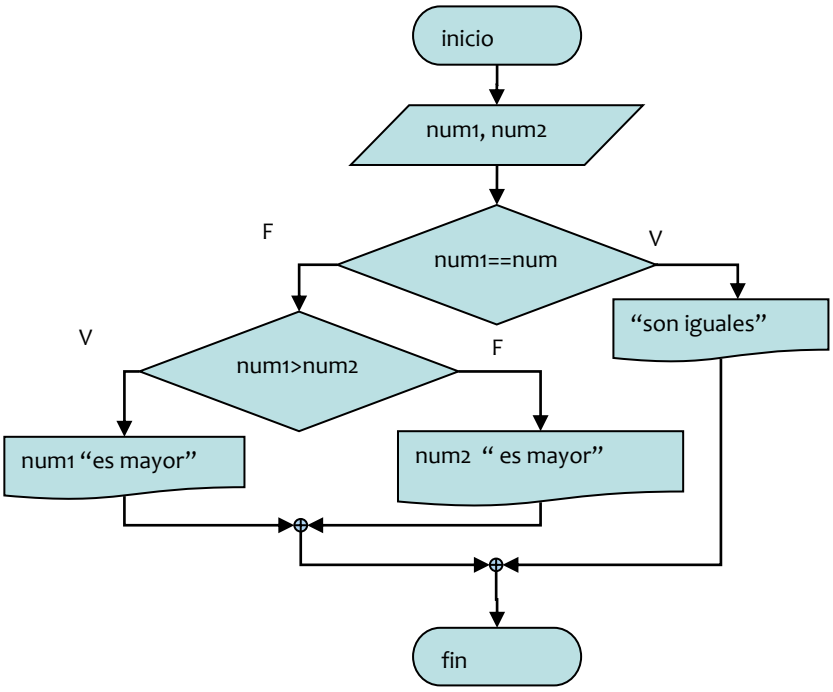


Figura II-35 *Determinar si hay un numero mayor o si son iguales.*

3.- Codificación. Completar el siguiente programa poniendo las condiciones que hacen falta y completando el `if` anidado.

```
// Determina si dos números son iguales o hay un mayor
// igualmayor.cpp
#include <iostream.h>

main() {
  int num1, num2;
  // Pedir los datos

  if( ... )
    cout << " los números son iguales ";
  else
    if( ... )

    else

  return 0;
}
```

A continuación presentamos el programa completo:

```
// Determina si dos números son iguales o hay un mayor
#include <iostream.h>

main() {
  int num1, num2;

  // Pedir los datos
  cout << "introduzca el primer número ";
  cin >> num1;
  cout << "introduzca el otro número ";
  cin >> num2;

  if( num1 == num2 )
    cout << " los números son iguales ";
  else
    if( num1 > num2 )
      cout << "el mayor es " << num1;
    else
      cout << "el mayor es " << num2;

  return 0;
}
```

Ejercicio 2.2.6: ordenar de mayor a menor 3 números diferentes entre si. Nótese que no se trata de decir cuál de los tres números es el mayor, sino de ordenarlos en forma descendente.

Solución:

1.- *Entradas:* sean A, B y C los tres números diferentes entre sí.

Salidas: será una de las seis permutaciones posibles:

A B C, A C B, C A B, B A C, B C A y C B A.

2.- En la Figura II-36 se muestra la forma en la que se obtendría la primera salida.

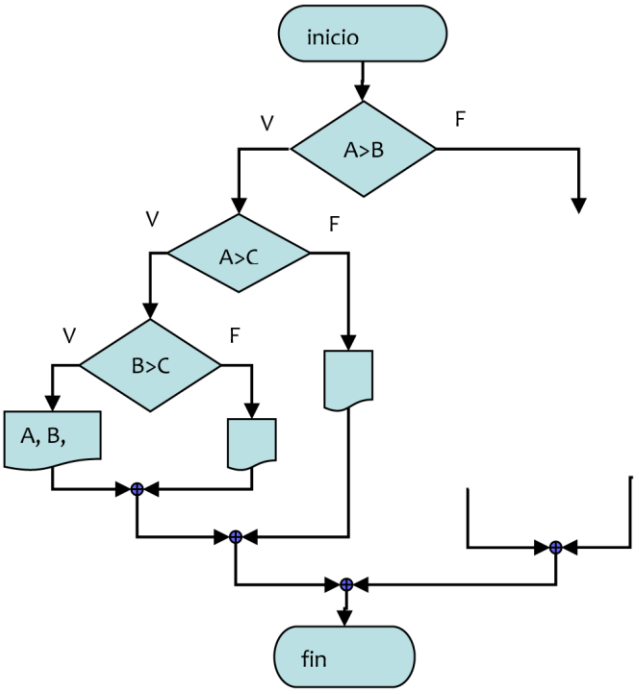


Figura II-36 *Ordenar 3 números de mayor a menor, primera opción.*

¿Cómo completaríamos todos los casos posibles para cuando A es mayor que B?

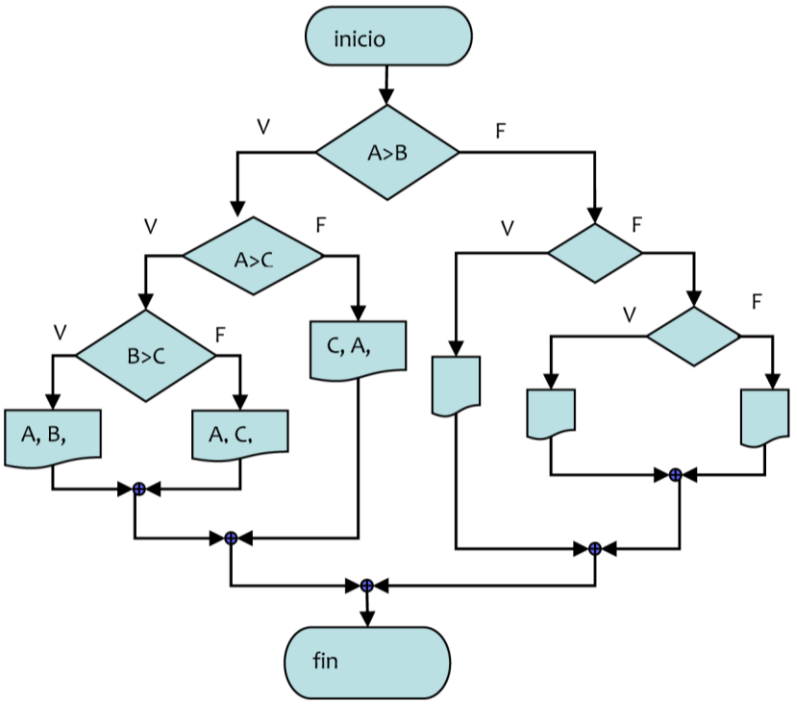


Figura II-37 *Ordenar 3 números de mayor a menor, A > B.*

En la Figura II-37 se muestran las tres posibles soluciones cuando A es mayor que B. El algoritmo completo consiste en tomar en cuenta también todas las posibilidades cuando B es mayor que A. ¿Puedes completarlo sin consultar la *Figura II-38.*

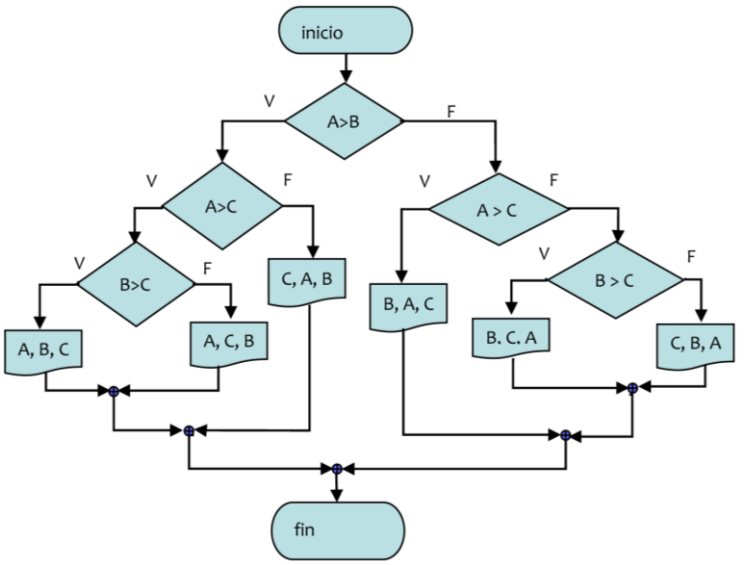


Figura II-38 *Ordenar 3 números de mayor a menor.*

3.- Codificación. En el siguiente programa se presenta únicamente dos niveles de anidamiento. ¿Cómo incluimos el tercer nivel de anidamiento?

```
// Ordena de mayor a menor tres números diferentes entre si
#include <iostream.h>

main() {
  int A, B, C;
  // pedir los datos

  if( A > B )
    if( A > C )
      .
    else
      cout << C << " " << A << " " << B;
  else
    if( A > C )
      cout << B << " " << A << " " << C;
    else
      ...
  return 0;
}
```

Aquí va un tercer `if` anidado.

Aquí va un tercer `if` anidado.

A continuación presentamos el programa completo:

```
// Ordena de mayor a menor tres números diferentes entre si
#include <iostream.h>

main() {
  int A, B, C;
  // Pedir los datos
  cout << "cuanto vale A?";  cin >> A;
  cout << "cuanto vale B?";  cin >> B;
  cout << "cuanto vale C?";  cin >> C;

  if( A > B )
    if( A > C )
      if( B > C )
        cout << A << " " << B << " " << C;
      else
        cout << A << " " << C << " " << B;
    else
      cout << C << " " << A << " " << B;
  else
    if( A > C )
      cout << B << " " << A << " " << C;
    else
      if( B > C )
        cout << B << " " << C << " " << A;
      else
        cout << C << " " << B << " " << A;

  return 0;
}
```

II.2.3 Estructura selectiva múltiple

En el caso de la estructura selectiva. sencilla, solo existen dos opciones, el flujo del programa se va por un camino cuando la condición es verdadera o por el otro camino cuando la condición es falsa. Ahora estudiaremos el caso en el que hay más de dos posibilidades a elegir. En este caso, es necesaria una estructura que permita que el flujo del programa se dIlida en varias ramas desde el punto de la *toma de decisión*, en función del valor que tome un *selector*.

El *selector* es una expresión que puede tomar varios valores, no solo dos. Cada uno de estos valores se asocia con una de las posibles ramas. Al final de la estructura, todas las ramas se unen en un mismo punto, como se ilustra en la Figura II-39.

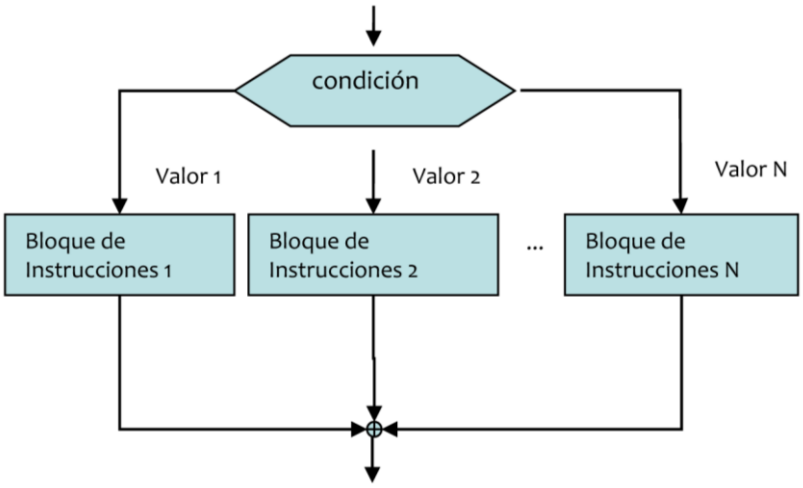


Figura II-39 *Estructura Selectiva múltiple.*

La instrucción `switch` de C++ permite que el programa evalúe un selector y luego elija el camino a seguir en base al valor de este selector. La sintaxis en C++ de la instrucción `switch` es la siguiente:

```
switch( selector )
{
  case <valor_1> : <instrucciones>
                   break;
  case <valor_2> : <instrucciones>
                   break;
  case <valor_N> : <instrucciones>
                   break;
  default: <instrucciones>   // opcional
};
```

Cuando un programa encuentra una instrucción *switch*, analiza primero la condición y luego intenta encontrar un valor coincidente dentro de los casos posibles. Al encontrarlo, ejecuta las instrucciones correspondientes.

La instrucción *break* indica que se termine la instrucción *switch* en curso y se prosiga la ejecución del programa en la primera instrucción que sigue al *switch*.

Si no se da la instrucción *break*, se seguirán ejecutando todas las instrucciones de los siguientes casos, por ejemplo, si son cinco casos y el selector tiene el valor para el caso 3, si no existe un *break* se ejecutarán las instrucciones correspondientes a los casos 3, 4 y 5.

Ejemplo 2.2.4.- Hacer un programa que de un mensaje que debe estar en función de la calificación que proporciona el usuario, la cual es un número entero que va del 5 al 10.

```
// Programa que da un mensaje en función de la calificación
// calific.cpp
#include <iostream.h>

main() {
  int calificacion;

  cout << " Cual fue tu calificacion? (un entero del 5 al 10)\n";
  cin >> calificacion;

  switch (calificacion) {
    case 10: cout<<" Felicidades por tu 10! \n ";
             break;
    case  9: cout<<" Bien por tu 9! \n ";
             break;
    case  8: cout<<" Un 8 no esta mal. \n ";
             break;
    case  7: cout<<" Abajo del promedio. \n ";
             break;
    case  6: cout<<" Panzaso! \n ";
             break;
    case  5: cout<<" Estudia para la próxima! \n ";
  };

  return 0;
}
```

Nótese que para el caso en el que la calificación es 5, el `break` ya no es necesario.

II.3 Estructuras repetitivas.

En muchos problemas de diseño de algoritmos es necesario indicar que uno o varios pasos deben ejecutarse en repetidas ocasiones, es decir, en un ciclo.

El número de repeticiones puede variar según las condiciones dadas al aplicar el algoritmo. Existen dos estructuras básicas para hacer que un bloque de instrucciones se repita: la *estructura repetitiva de control previo* y la *estructura repetitiva de control posterior*.

II.3.1 Estructura repetitiva de control previo.

La *estructura repetitiva de control previo* hace que un bloque de instrucciones se ejecute mientras sea verdadera (V) una condición dada, y termina cuando al condición es falsa (F). Esta estructura se muestra en la Figura II-40.

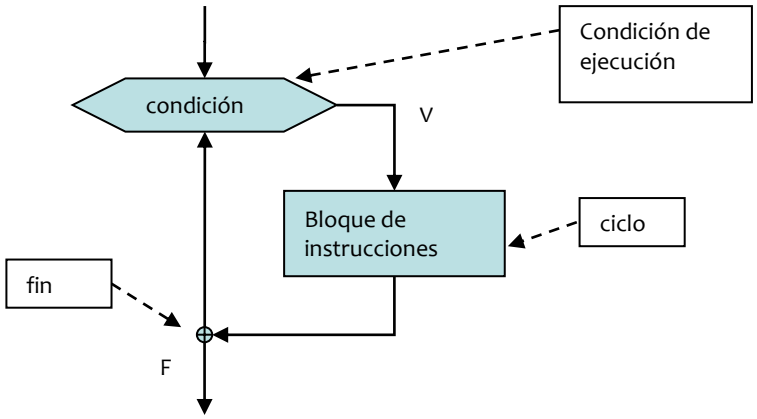


Figura II-40 *Estructura repetitiva de control previo.*

Esta estructura consta de tres partes:

1.- Condición de ejecución: Se evalúa para decidir si se debe ejecutar una nueva iteración o terminar el ciclo.

2.- Ciclo: Es el bloque de instrucciones que se ejecutarán repetidamente.

3.- Fin: marca el punto en el que la ejecución continúa cuando la condición de ejecución resulta falsa.

Nota: es muy importante que exista una instrucción dentro del ciclo que pueda, en algún momento, hacer que la condición sea falsa. Esto es para evitar que el ciclo se ejecute indefinidamente.

La estructura *estructura repetitiva de control previo* se representa en C++ con la instrucción `while` que en español significa *mientras*. En esta estructura el ciclo se ejecuta *mientras la condición sea verdadera.* La sintaxis del `while` es la siguiente:

```
while( condición ) {

  <bloque de instrucciones>

};.
```

Ejemplo 2.3.1.- diseñar un algoritmo para que la computadora escriba los números naturales menores que 100.

Solución:

1.- *Determinar entradas y salidas*.- En este caso no hay entradas, solo un contador al que llamaremos `n` y al que daremos un valor inicial de 1. Las salidas serán todos los números enteros del 1 al 99.

2.- Especificar las operaciones. El contador n se despliega en la pantalla y se va incrementando de 1 en 1. En la Figura II-41 se muestra el algoritmo que escribe los números naturales del 1 al 99.

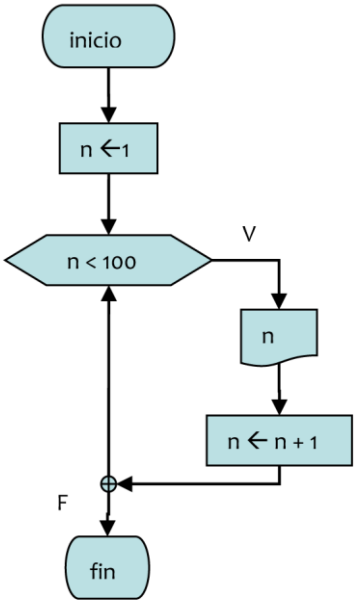


Figura II-41 *Escribiendo del 1 al 99 de 1 en 1.*

3.- *Codificación*: a continuación el programa en C++ del algoritmo anterior.

```
// Escribe los números naturales menores
// Que 100
#include <iostream.h>

main() {
  int n = 1;

  while( n < 100 ) {
    cout << n << " ";
    n++;
  };

  return 0;
}
```

El ciclo se repite mientras que `n` sea menor que `100`, es, decir, hasta que `n = 99`

`n` se incrementa de `1` en `1`

Ejemplo 2.3.2.- La $n-ésima$ potencia factorial de un número entero x se define como:

$$x^{(n)} = x(x-1)(x-2)....(x-n+1)$$

De tal manera que cada factor va disminuyendo hasta que el último factor es $(x-n+1)$. Si $x<n$ la $n-ésima$ potencia factorial de x es cero. A continuación se dan algunos ejemplos:

$$5^{(2)} = 5(5-1) = 5*4 = 20$$

$$4^{(3)} = 4(4-1)(4-2) = 4*3*2 = 24$$

$$2^{(3)} = 2(2-1)(2-2) = 2*1*0$$

Hacer el diagrama de flujo y el programa en C++ para calcular la $n-ésima$ potencia factorial de x.

Entradas:

La base: `x` La potencial factorial: `n`

Salidas:

La potencia factorial: `potFact`

Otras variables: Usaremos la variable `factor` que tendrá un valor inicial igual a `x` y se irá decrementando hasta que sea igual a `x-n+1`.

Lo primero es pedir los datos y cerciorarse de que la potencia sea menor o igual que la base, en caso contrario, el resultado es cero.

Solución:

1.- *Determinar entradas y salidas*.- La base, que en este caso será `x`, y la potencia factorial, que será `n`. Para la salida será `potFact`. Ver Figura II-42.

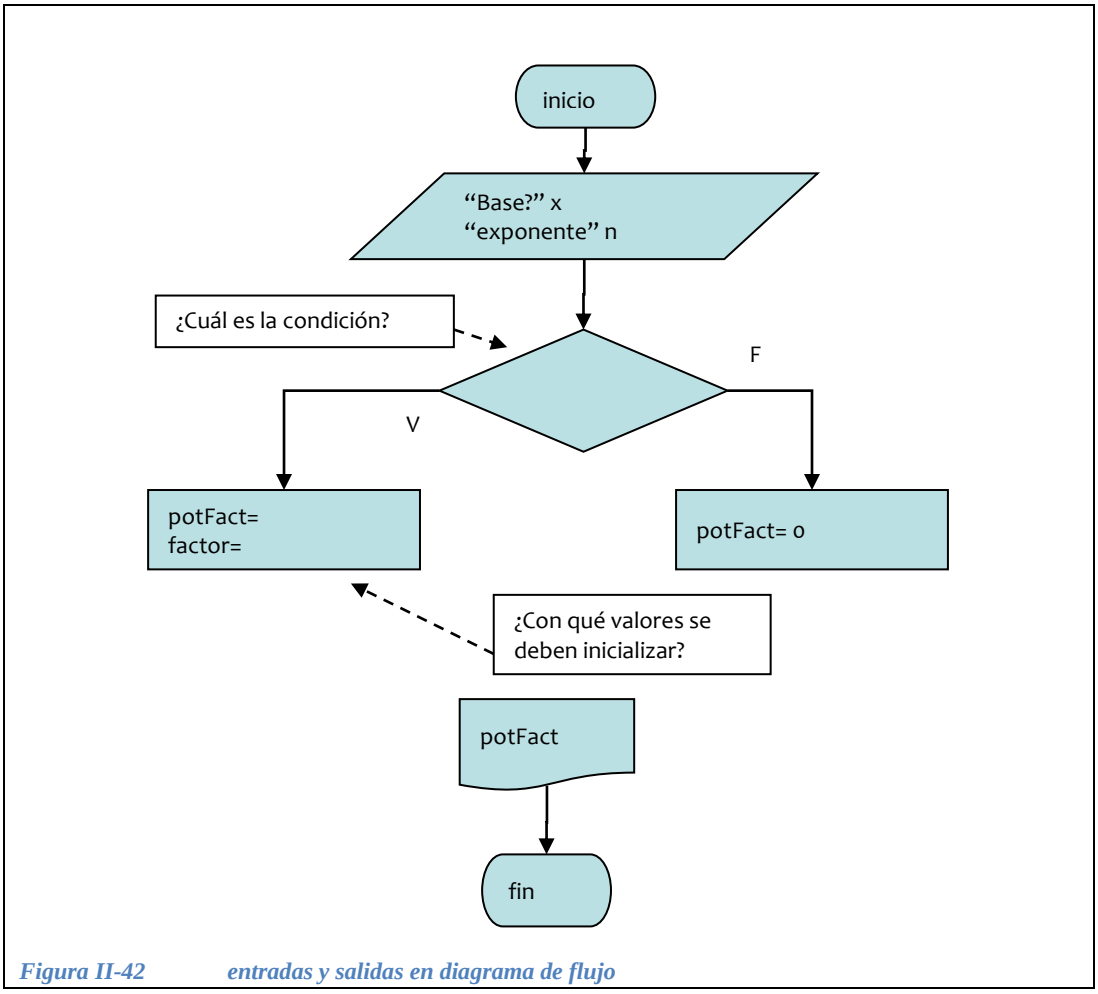


Figura II-42 *entradas y salidas en diagrama de flujo*

2.- *Determinar el flujo principal*. ¿qué condición se debe cumplir para decidir si el resultado es cero o no? ¿con qué valores se deben inicializar las variables en caso de que el resultado no sea cero? Ver solución en la Figura II-43.

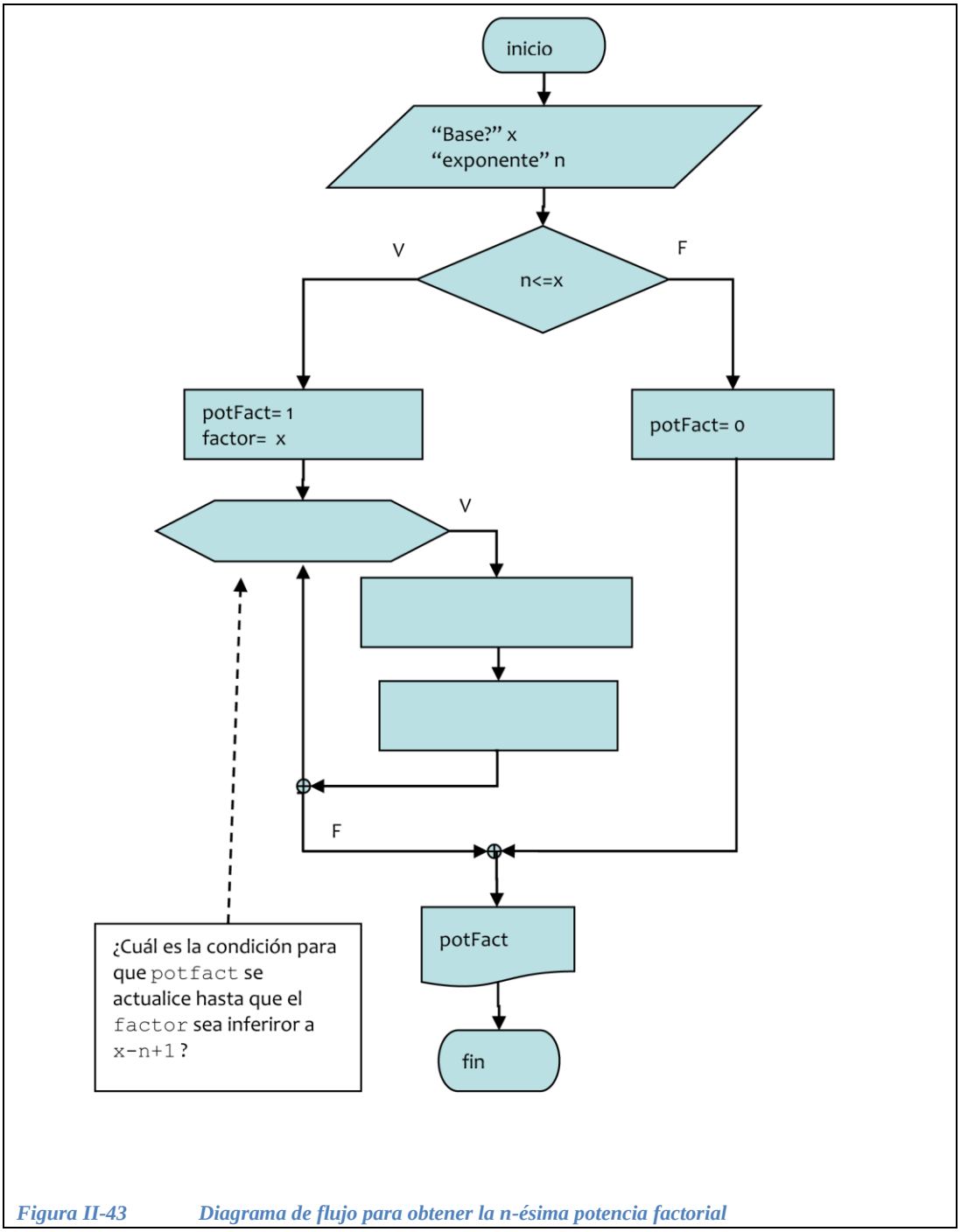


Figura II-43 *Diagrama de flujo para obtener la n-ésima potencia factorial*

3.- *Especificar las operaciones*. Completar el diagrama de la Figura II-43 con los pasos necesarios para calcular la potencia factorial (ver solución en laFigura II-44).

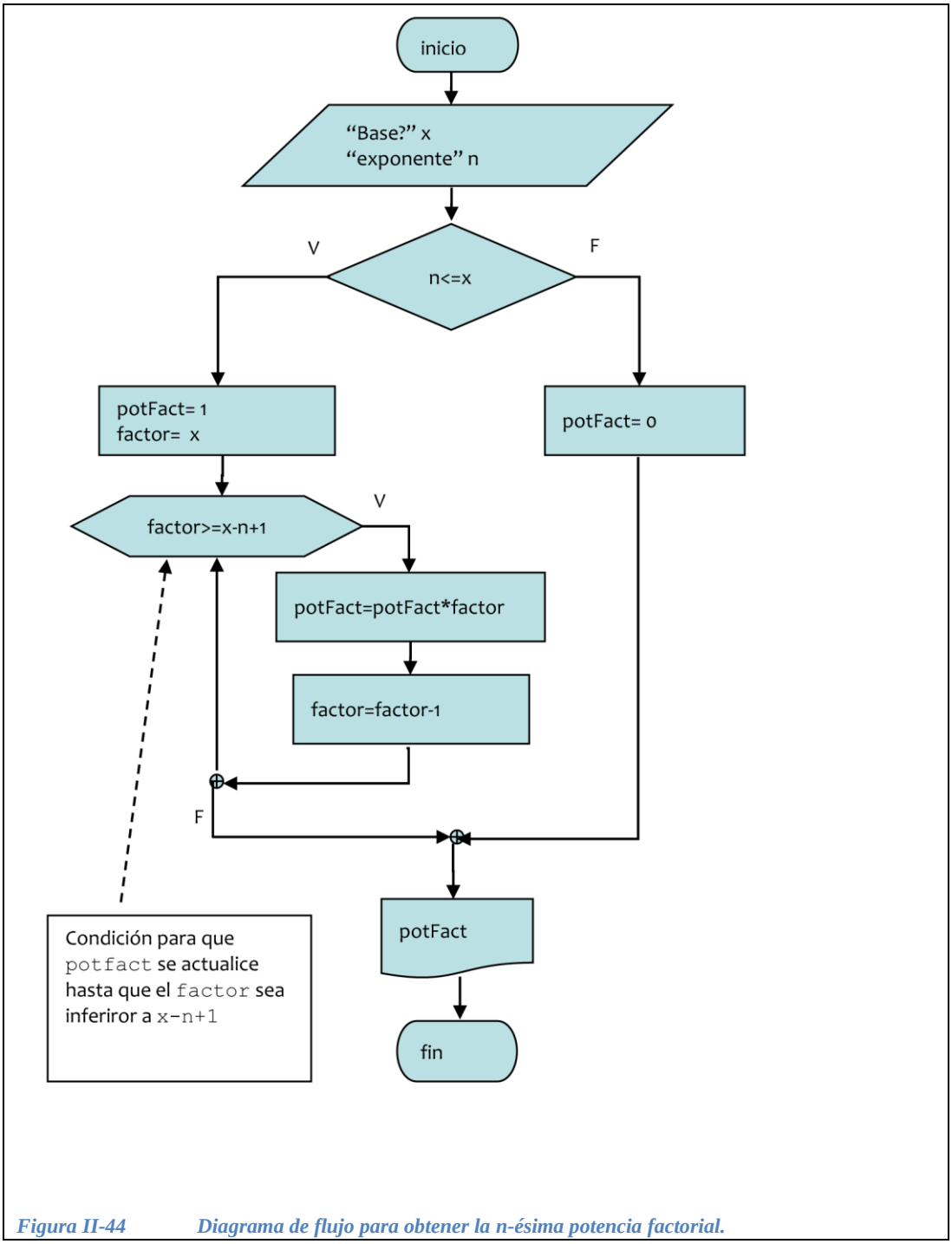
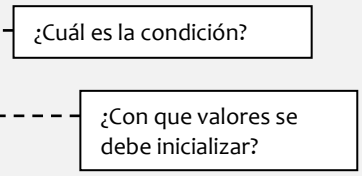


Figura II-44 *Diagrama de flujo para obtener la n-ésima potencia factorial.*

4.- *Codificación*: a continuación se tiene la primera parte del programa en C++ del algoritmo anterior.

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

int main() {
  int x, n, factor, potFact;

  cout<<"Cual es el valor de la base?:   ";
  cin>>x;
  cout<<"Cual es el exponente?:  ";
  cin>>n;
  if(         ) {
    potFact=?;
    factor=?;
    while(        ){

    }
  }
  else
    potFact = 0;

  cout<<"La potencia factorial: "<< n << " de " << x << " es: "
      << potFact <<endl;

  system("PAUSE");
  return 0;
}
```



5.- *Codificación*. Segunda parte de la *solución parcial* en C++.

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

int main() {
  int x, n, factor, potFact;

  cout<<"Cual es el valor de la base?:   ";
  cin>>x;
  cout<<"Cual es el exponente?:  ";
  cin>>n;
  if( n <= x ) {
    potFact=1;
    factor=x;
    while(                ) {
          <------------
    }
  }
  else
    potFact = 0;

  cout<<"La potencia factorial: "<< n << " de " << x << " es: "
      << potFact <<endl;

  system("PAUSE");
  return 0;
}
```

¿Cuál es la condición?

¿Cuáles son las operaciones que deben realizarse?

5.- A continuación se muestra el programa completo:

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

int main() {
  int x, n, factor, potFact;

  cout<<"Cual es el valor de la base?:   ";
  cin>>x;
  cout<<"Cual es el exponente?:  ";
  cin>>n;
  if( n <= x ) {
    potFact=1;
    factor=x;
    while( factor >= x-n+1 ) {
      potFact = potFact*factor;
      factor = factor-1;
    }
  }
  else
    potFact = 0;

  cout<<"La potencia factorial: "<< n << " de " << x << " es: "
      << potFact <<endl;

  system("PAUSE");
  return 0;
}
```

II.3.2 Estructura repetitiva de control posterior

La *estructura repetitiva de control posterior* hace que la ejecución de un bloque de instrucciones se repita mientras sea verdadera una condición dada.

Esta estructura es parecida a la repetitiva de control previo, sin embargo sus partes están en otro orden y su operación es consecuentemente distinta. Esta estructura se muestra en la Figura II-45.

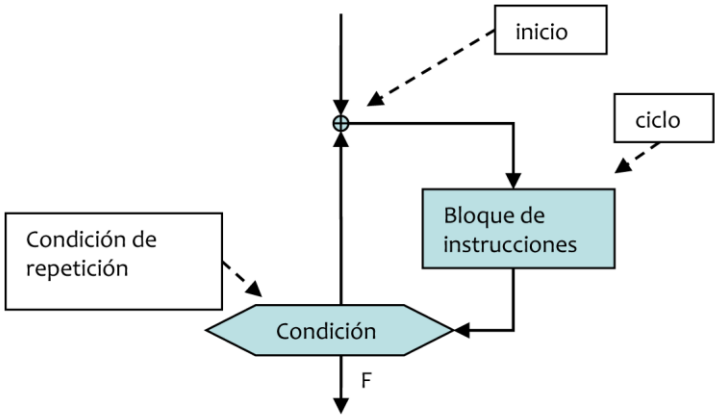


Figura II-45 *Estructura repetitiva de control posterior.*

La estructura repetitiva de control posterior consta de las siguientes partes:

1.- Inicio: marca el punto donde la ejecución continúa en caso de que la condición de repetición resulte verdadera

2.- Ciclo: es el bloque de instrucciones que se ejecutan y posiblemente se repiten

3.- Condición de repetición: se evalúa para decidir si el bloque de instrucciones debe repetirse.

Las instrucciones del ciclo se ejecutan de inmediato, luego se evalúa la condición de repetición. Si es verdadera (V), se repite la ejecución del ciclo para luego volver a evaluar la condición y así sucesivamente. Cuando la condición de repetición se vuelve falsa (F), el programa prosigue su ejecución en la primera instrucción que se encuentre después de la estructura

La *estructura repetitiva de control posterior* se representa en C++ con la instrucción `do-while` que en español significa *hacer mientras*. En esta estructura el ciclo se ejecuta *mientras la condición sea verdadera*, sin embargo, aunque la condición siempre sea falsa, el ciclo se ejecuta por lo menos una vez. La sintaxis del `do-while` es la siguiente:

```
do {

  <bloque de instrucciones>

}while( condición );
```

Nota: La *estructura repetitiva de control posterior* se utiliza cuando se requiere que el ciclo se ejecute por lo menos una vez y luego, basándose en cierta condición, tal vez repetirlo.

Ejemplo 2.3.3.- Diseñar un algoritmo para que la computadora sume 100 números diferentes que proporciona el usuario.

Solución:

1.- *Determinar entradas y salidas*: al diferencia del ejemplo 2.3.1, el contador `n` solo lo utilizaremos para pedir los 100 números, las entradas serán los 100 números que proporcione al usuario. Pediremos de una en una y la guardaremos en la variable `x`, la salida será la sumatoria de los 100 números, y le llamaremos `suma`.

2.- *Especificar las operaciones*: en este caso utilizaremos lo que se llama un ***acumulador***. Un acumulador, como su nombre lo indica, sirve para ir acumulando resultados parciales, de la siguiente manera:

Por ejemplo:

```
acumulador ← acumulador + lo que se desea acumular
suma ← suma + x
```

Un acumulador debe *inicializarse* antes de usarse, para evitar acumular basura, para el caso de la suma, el acumulador debe inicializarse con cero.

```
suma ← 0
```

El diagrama de flujo que pide 100 números al usuario, obtiene la sumatoria de estos 100 números y la muestra en pantalla se ilustra en la Figura II-46.

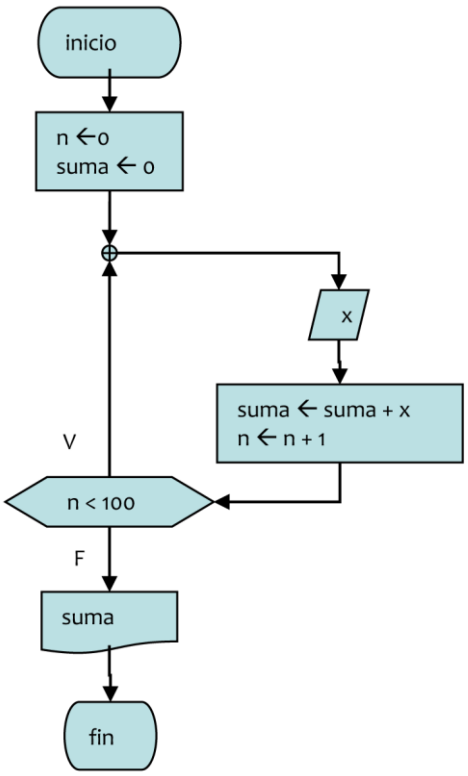


Figura II-46 *Pide 100 números al usuario y obtiene la sumatoria.*

3.- Codificación:

```
// Obtiene la suma de 100 números
#include <iostream.h>

main() {
  int n = 0;
  float suma = 0, x;

  cout << "escribir 100 números" << endl;
  do {
    cin >> x;
    suma = suma + x;
    n++;
  }while( n < 100 );
  cout << "la suma es " << suma << endl;

  return 0;
}
```

II.3.3 Estructura repetitiva de control de contador

Esta estructura se utiliza normalmente cuando el control de un ciclo se hace mediante un *contador.* Esta estructura se muestra en la Figura II-47.

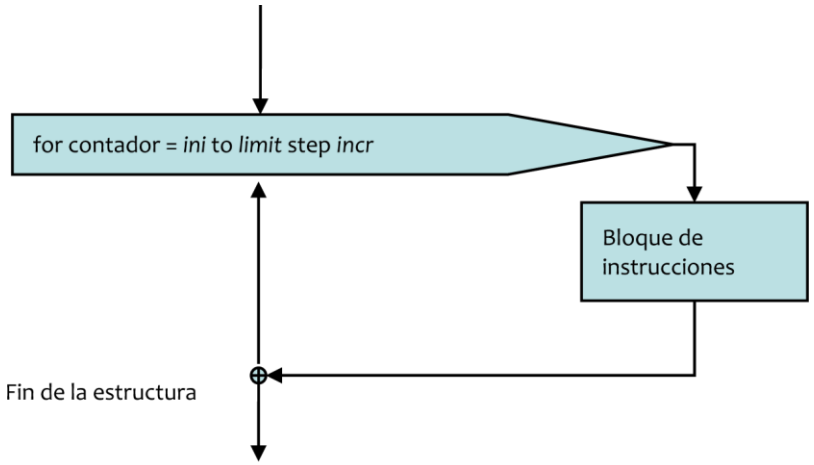


Figura II-47 *Estructura repetitiva de control de contador.*

El bloque de instrucciones se ejecuta una vez para cada valor del *contador* iniciando con el valor *ini*, este valor se en cada iteración con el valor *incr* hasta llegar a *limit*.

La estructura **repetitiva de control de contador** está definida en C y C++ de la siguiente manera:

```
for( <inicialización>; <condición>; incremento ) {
  <bloque de instrucciones>;
};
```

Aquí nunca va ";"

En C y C++ esta estructura es realmente un caso particular de la **estructura repetitiva de control previo** orientado a contar las repeticiones. Lo que se pretende con la estructura **repetitiva de control de contador** es disminuir la probabilidad de cometer algunos errores, como:

- No inicializar correctamente las variables.
- Tener un ciclo infinito.

Las cuatro partes del **for** en C y C++ **en el orden de su ejecución** son las siguientes:

- Inicialización:

  Se asigna el valor inicial al contador del bucle o ciclo.
  Esta parte se ejecuta una sola vez.

- Condición de iteración:

  Es una expresión lógica. Mientras ésta sea verdadera se ejecuta una iteración más del ciclo.

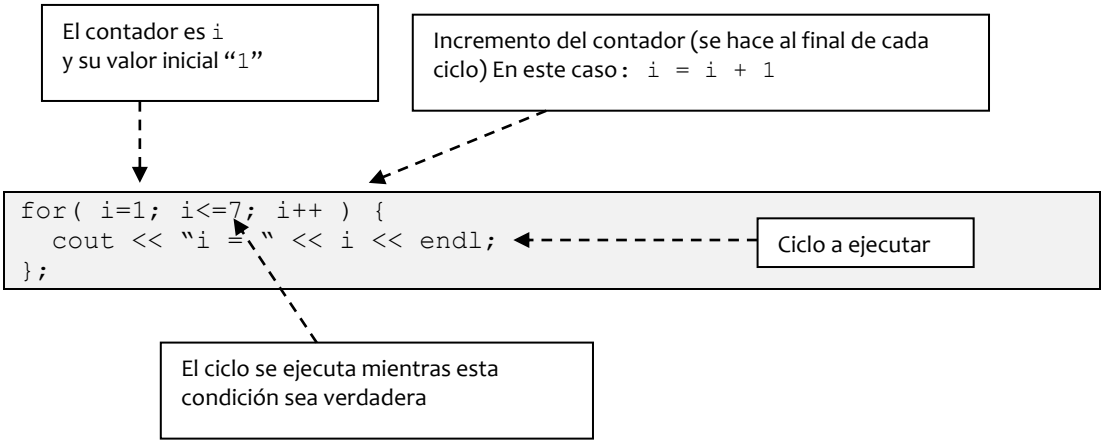
- Ciclo o bucle:

Es el conjunto de instrucciones que se ejecutan en caso de que la condición de iteración sea verdadera.

- Incremento:

Incrementa o decrementa el contador del ciclo. Se ejecuta después de cada iteración del ciclo y la ejecución continúa con la condición de iteración.

Ejemplo:



Es una buena costumbre de programación declarar las variables lo más cerca del bloque donde se van a utilizar, por esto se recomienda declarar el contador en la sección de inicialización del ciclo **for**:

Ejemplos:

```
for( int i=0; i<=32; i=i+7 )
  cout << " i = " << i << ", ";
for( long int j=100000L; j>0; j=j-5 )
  cout << " j = " << j << endl;
```

Al igual que en **if** y **while,** los corchetes no son necesarios cuando hay una sola instrucción.

Ejemplo 2.3.4.- Hacer un programa que cuente en la pantalla del cero al 1000 de dos en dos.

Solución:

1.- *Determinar entradas y salidas*. En este caso no hay entradas, como salida tendremos al contador `i` que cuenta de dos en dos del 0 al 1000. Hay que inicializar `i=0`.

2.- *Especificar las operaciones*, el algoritmo para resolver este problema se muestra en el diagrama de flujo de la Figura II-48.

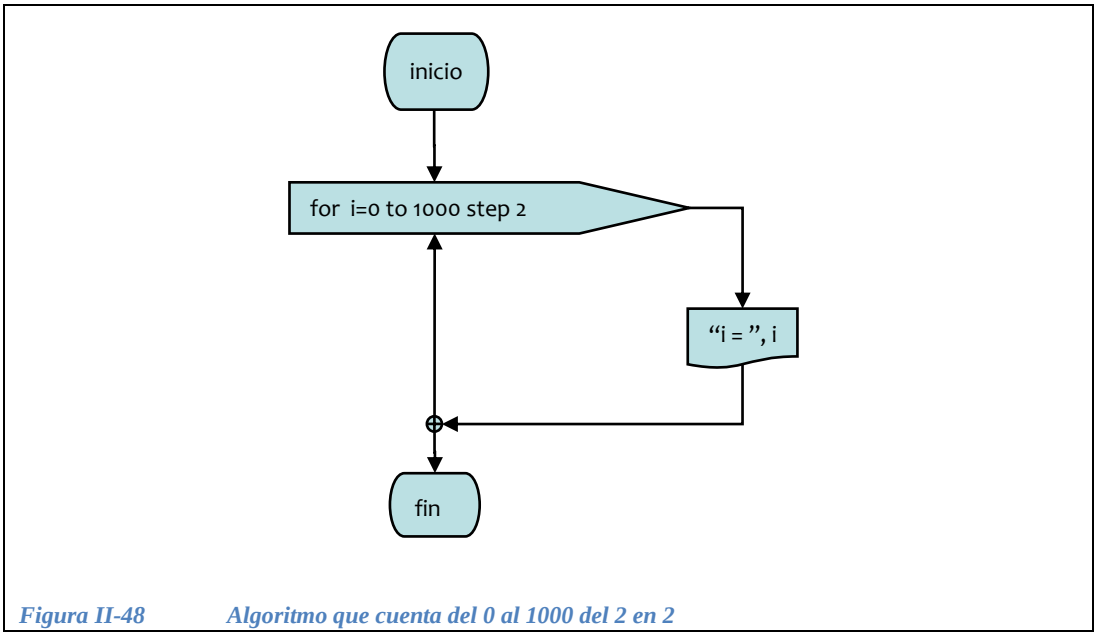


Figura II-48 Algoritmo que cuenta del 0 al 1000 del 2 en 2

3.- *Codificación*. A continuación se muestra el código en C++ del algoritmo anterior:

```
// Cuenta de 2 en 2 del 0 al 1000
// cuenta0_1000.cpp

#include <iostream.h>

main() {
  for( int i=0; i<=1000; i=i+2 )
    cout << " i = " << i << ", ";

  return 0;
}
```

Usamos <= cuando incluimos el límite

Ejemplo 2.3.5.- Hacer un programa que cuente en la pantalla del 40 al 10 de 5 en 5.

Solución:

1.- *Determinar entradas y salidas*. En este caso no hay entradas, como salida tendremos al contador `i` que cuenta de dos en dos del 0 al 1000. Hay que inicializar `i = 40.`

2.- *Especificar las operaciones*, el algoritmo para resolver este problema se muestra en el diagrama de flujo de la Figura II-49.

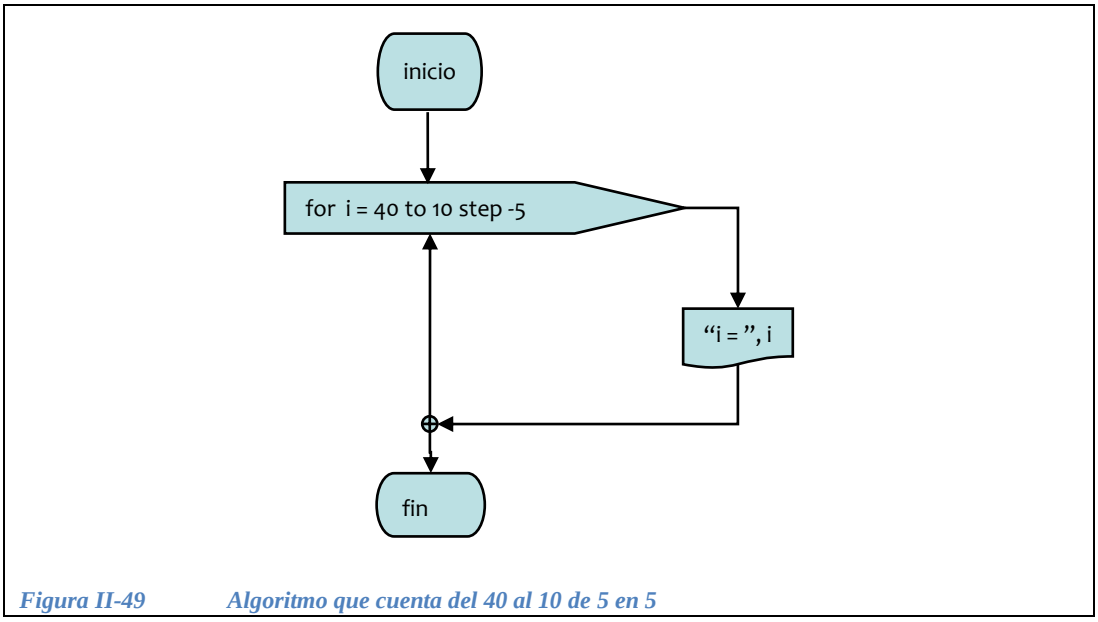


Figura II-49 Algoritmo que cuenta del 40 al 10 de 5 en 5

3.- *Codificación*. A continuación se muestra el código en C++ del algoritmo anterior:

```
// Cuenta de 5 en 5 del 40 al 10
// cuenta40_10.cpp

#include <iostream.h>

main() {
  for( int i=40; i > 9; i=i-5 )
    cout << " i = " << i << ", ";

  return 0;
}
```

Usamos > cuando NO incluimos el límite

Ejemplo 2.3.6.- Hacer un programa que obtenga el promedio de `n` números utilizando la estructura de control de contador.

Solución:

1.- *Determinar entradas y salidas*. En este caso las entradas son el número de números: `n` y los `n` números que introduce el usuario, a los que llamaremos: `num`, como salida tendremos el promedio de los `n` números: `promedio`.

2.- *Especificar las operaciones*, Utilizaremos un acumulador `suma` para ir obteniendo la sumatoria de todos los números que introduzca el usuario, es necesario inicializar el acumulador `suma = 0`, el promedio se obtiene dividiendo la suma total entre el número `n`.

El algoritmo para resolver este problema se muestra en el diagrama de flujo de la Figura II-50.

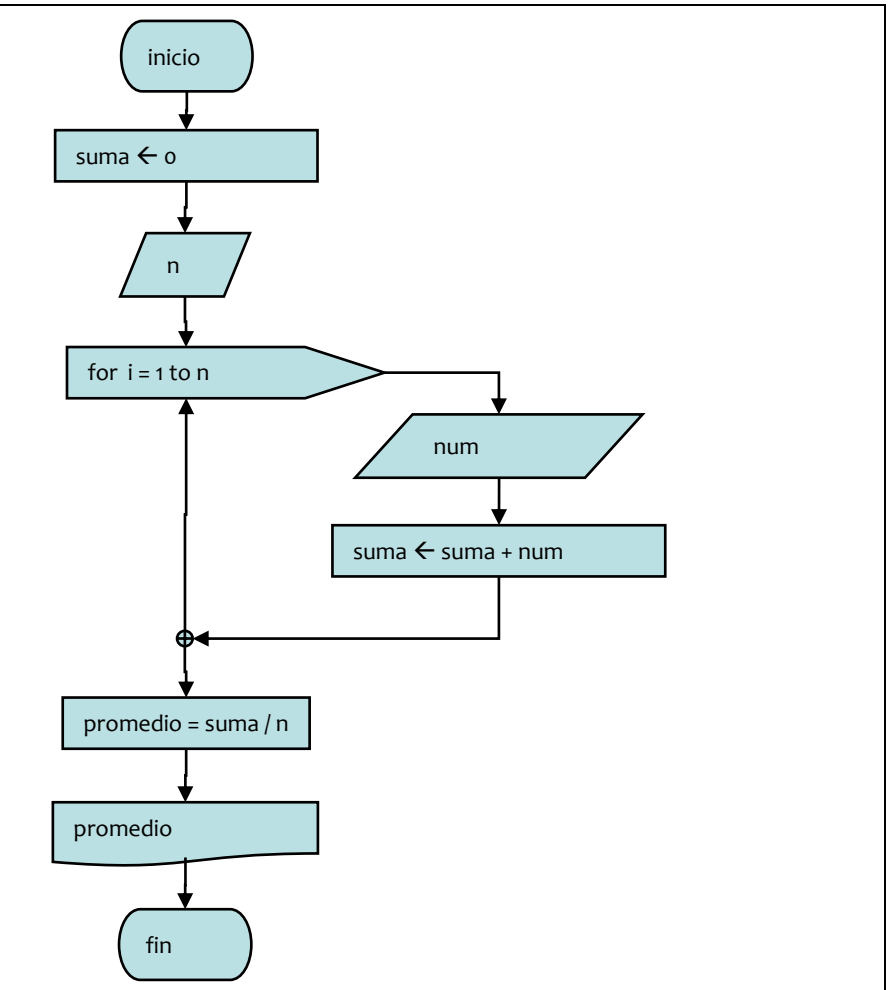


Figura II-50 *Algoritmo que obtiene el promedio de n números*

3.- *Codificación*. A continuación se muestra el código en C++ del algoritmo anterior:

```
// Obtiene el promedio de n números
// promedio.cpp
#include <iostream.h>

float num, suma=0, promedio;
int   n;

main() {
  cout << "cuantos numeros?";
  cin >> n;
  for( int i=0; i<n; i++ ) {
    cout << "introduce el número " << i+1 << endl;
    cin >> num;
    suma = suma + num;
  };

  promedio = suma/n;
  cout << "el promedio es "<< promedio;
  return 0;
}
```

4.- Para una mejor comprensión del funcionamiento de este algoritmo, se incluye la siguiente prueba de escritorio:

N	num	suma	i	suma (siguiente valor)
5	15	0	1	0 + 15 = 15
	7	15	2	15 + 7 = 22
	2	22	3	22 + 2 = 24
	3	24	4	24 + 3 = 27
	5	27	5	27 + 5 = 32

Ejemplo 2.3.7.- Obtener el *Factorial* con estructura repetitiva de control de contador .

El acumulador también puede usarse en una multiplicatoria en lugar de una sumatoria:

Para la sumatoria:

acumulador = **acumulador** + *lo que se desea acumular*

en el ejemplo anterior:

`suma = suma + x;` e inicializamos `suma = 0`

Para la multiplicatoria:

acumulador = **acumulador** * *lo que se desea acumular*

Por ejemplo:

```
factorial = factorial * i;
```

1.- *Determinar entradas y salidas*. En este caso la entrada es el número del que obtendremos su factorial: `n`, como salida tendremos el factorial del número `n`.

2.- *Especificar las operaciones*, Utilizaremos el acumulador `factorial` para ir obteniendo la multiplicatoria, es necesario inicializar el acumulador `factorial = 1`.

El algoritmo para resolver este problema se muestra en el diagrama de flujo de la Figura II-51.

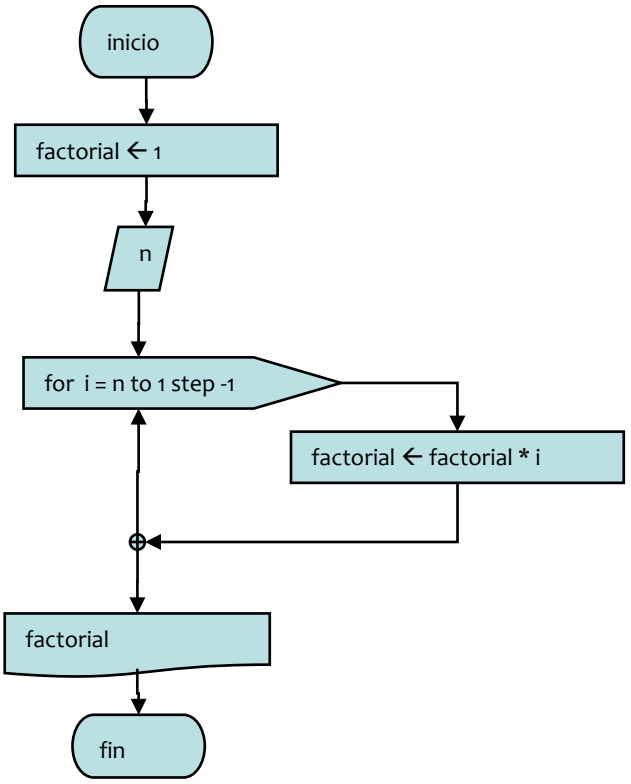


Figura II-51 Algoritmo para obtener el factorial del número n.

3.- *Codificación*. A continuación se muestra el código en C++ del algoritmo anterior:

```
// Programa que obtiene el
// Factorial de un número fact.cpp
#include <iostream.h>

long int fact = 1;
int n;

main() {
  cout << "que número? ";
  cin >> n;

  for( int i=n; i>1; i-- )
    fact = fact * i;

  cout << "El factorial de "<< n
       << "es: " << fact;

  return 0;
}
```

4.- Para una mejor comprensión del funcionamiento de este algoritmo, se incluye la prueba de escritorio siguiente:

n	fact	i	fact (siguiente valor)
5	1	5	1 * 5 = 5
	5	4	5 * 4 = 20
	20	3	20 * 3 = 60
	60	2	60 *2 = 120

Nótese que el código esta optimizado para que el ciclo ya no se ejecute cuando `i = 1`, ya que no tiene caso la multiplicación por uno.

II.3.4 Ejercicios de estructuras repetitivas

Ejercicio 2.3.1.- Diseñar un algoritmo para que la computadora escriba en pantalla uno a uno los elementos de la serie

- 7, 11, 15, 19, 23 ... mientras éstos sean menores a 100,000.
- 5/n, con n=1,2,3 ... mientras éstos sean mayores a 10^{-10}.

Solución a):

Ver figura II-52.

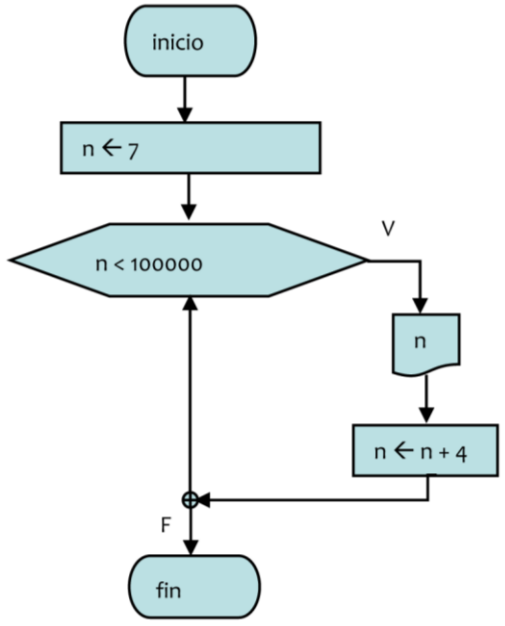


Figura II-52 *Algoritmo para escribir la serie 7, 11, 15, 19, … mientras los números sean menores a 100,000.*

Código:

```
// escribe los elementos de la serie
// 7, 11, 15, 19,… menores a 100000
#include <iostream.h>

main() {
  long int n = 7;
  while( n < 100000L ) {
    cout << n << " ";
    n = n + 4;
  };

  return 0;
}
```

se usa long porque int no alcanza

la `L` indica que la cantidad es de tipo `long int`

Solución b):

Ver Figura II-53.

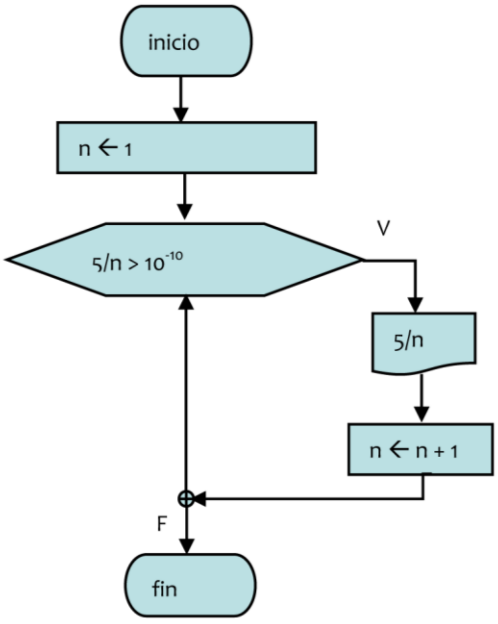


Figura II-53 *Algoritmo para escribir 5/n para n=1, 2, 3, 4, … mientras estos números sean mayores a 10^{-10}.*

Código:

```
// escribe los elementos de la serie 5/n
// mayores que 1e-10
#include <iostream.h>

main() {
  double n = 1;
  while( 5.0/n > 1e-10 ) {
    cout << 5.0/n << endl;
    n = n + 1;
  };
  return 0;
}
```

`float` no es suficiente porque solo tiene 7 dígitos de precisión y se requieren 11 para poder hacer `n = n + 1;` y llegar hasta `5e10`.
`double` tiene 15 dígitos de precisión

Ejercicio 2.3.2..- Diseñar un algoritmo para que la computadora escriba en un archivo los primeros 200 elementos de la serie

- 7, 11, 15, 19, 23, …
- 1000, 800, 600, 400, …
- 1, 2, 4, 8, 16, 32, 64, …

Solución a):

Ver Figura II-54.

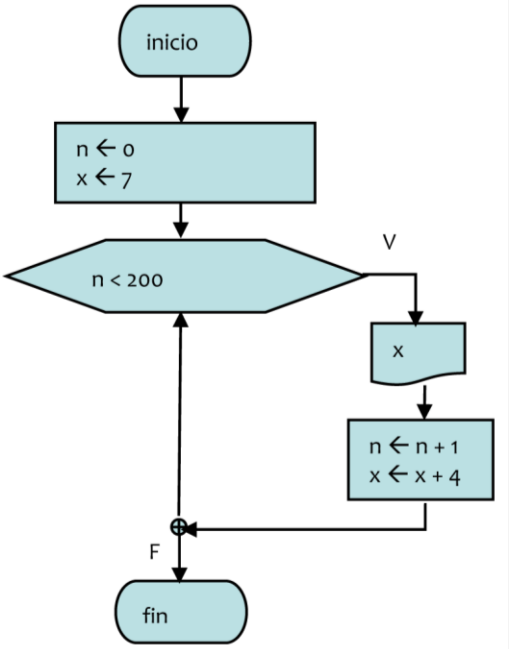


Figura II-54 *Algoritmo para escribir los primeros 200 elementos de la serie 7, 11, 15, 19, …*

Código:

```
// escribe los primeros 200 elementos de la
// serie 7, 11, 15, 19, …
#include <fstream.h>

main() {
  int n = 0;
  int x = 7;
  ofstream salida;
  salida.open("serie.txt");

  while( n < 200 ) {
    salida << x << " ";
    n = n + 1;
    x = x + 4;
  };

  salida.close();
  return 0;
}
```

Solución b): Ver Figura II-55.

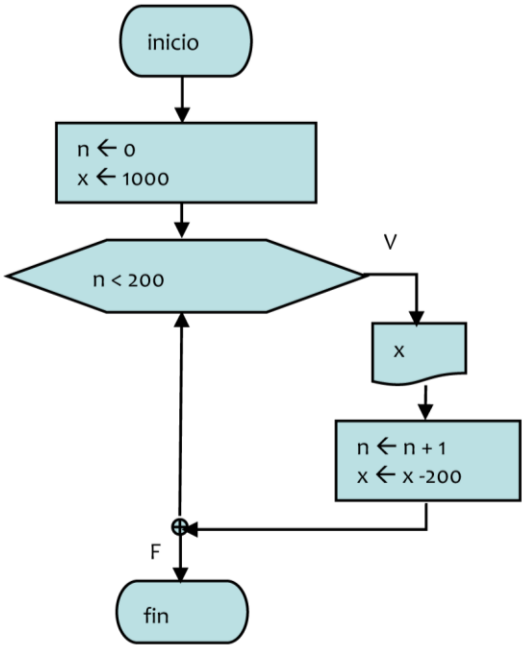


Figura II-55 *Algoritmo para escribir los primeros 200 elementos de la serie 1000, 800, 600, …*

Código:

```
// escribe los primeros 200 elementos de la
// serie 1000, 800, 600, 400, …
#include <fstream.h>

main() {
  int n = 0;
  long int x = 1000;
  ofstream salida;
  salida.open("serie.txt");

  while( n < 200 ) {
    salida << x << " ";
    n = n + 1;
    x = x - 200;
  };

  salida.close();
  return 0;
}
```

Solución c) Ver Figura II-56.

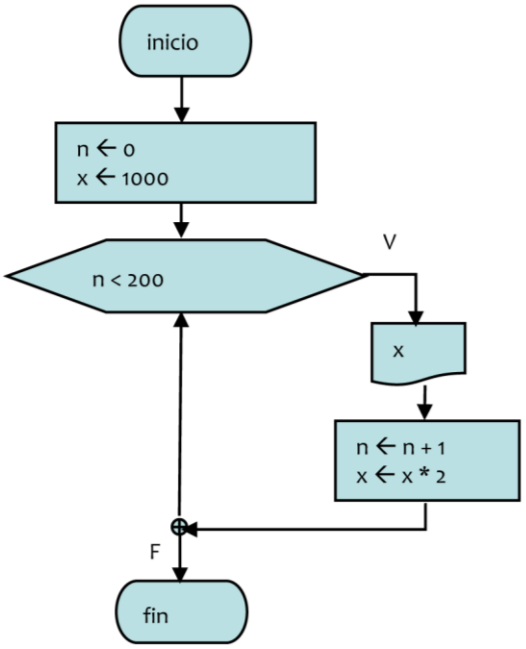


Figura II-56 *Algoritmo para escribir los primeros 200 elementos de la serie 1, 2, 4, 8, 16, …*

Código:

```
// escribe los primeros 200 elementos de la
// serie 1, 2, 4, 8, 16, 32, 64, …
#include <fstream.h>

main() {
  int n = 0;
  double x = 1;
  ofstream salida;
  salida.open("serie.txt");

  while( n < 200 ) {
    salida << x << " ";
    n = n + 1;
    x = x * 2;
  };

  salida.close();
  return 0;
}
```

Ejercicio 2.3.3.- Diseñar un algoritmo para que la computadora sume los montos de los gastos realizados por una persona mientras éstos sean mayores a cero.

- Se sumarán las cantidades mayores a cero solamente.
- Cuando se introduzca una cantidad menor a cero, la computadora deberá escribir la suma en un archivo de disco.

Solución: ver Figura II-57

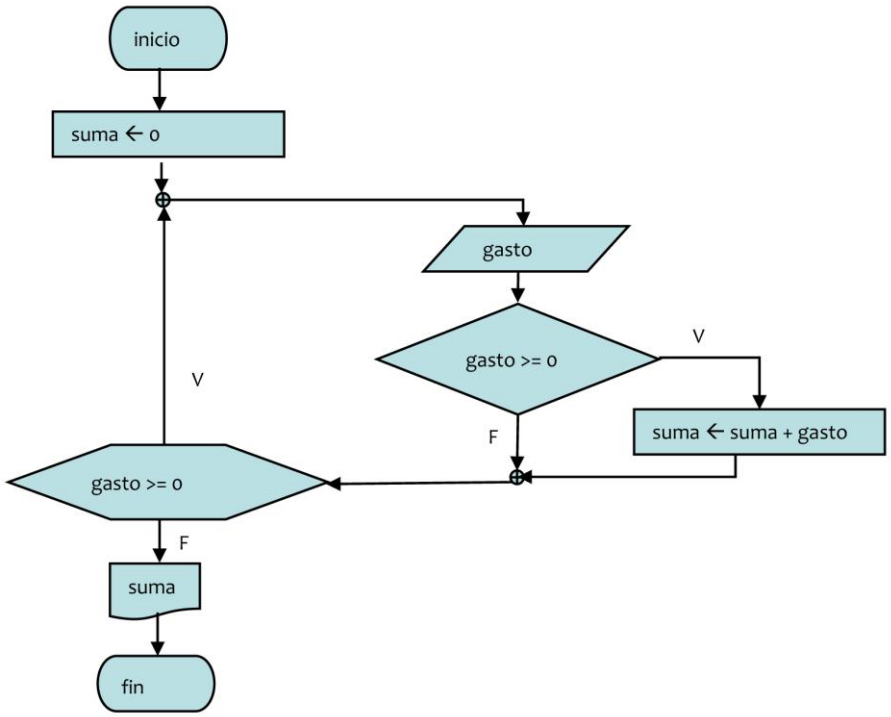


Figura II-57 *Algoritmo para capturar gastos y sumarlos hasta que se capture un gasto negativo o cero.*

Código:

```
// captura gastos
#include <fstream.h>
#include <iostream.h>

main() {
  double suma = 0, gasto;

  do {
    cout << "dar un monto ";
    cin >> gasto;
    if( gasto >= 0.0 )
      suma = suma + gasto;
  } while( gasto >= 0.0 );

  ofstream salida;
  salida.open("TotalGastos.txt");
  salida << suma;
  salida.close();

  return 0;
}
```

Ejercicio 2.3.4.- Modificar el algoritmo anterior para que primero lea del archivo la suma y que a ésta le agregue los montos adicionales que se capturen. Al final deberá substituir en el archivo la suma por la cantidad nueva.

Solución:

```
// captura gastos
#include <fstream.h>
#include <iostream.h>

main() {
  double suma, gasto;

  ifstream entrada;
  entrada.open("Total.txt");
  entrada >> suma;
  entrada.close();

  do {
    cout << "dar un monto ";
    cin >> gasto;
    if( gasto >= 0.0 )
      suma = suma + gasto;
  } while( gasto >= 0.0 );

  ofstream salida;
  salida.open("Total.txt");
  salida << suma;
  salida.close();

  return 0;
}
```

II.4 Construcción de menús con estructuras de control

Uno de los usos más comunes de la estructura selectiva múltiple, es en la construcción de menús. Un menú permite elegir al usuario entre una serie de opciones. Antes de continuar con el siguiente ejemplo, recordemos el *tipo enumerado* enum, visto en el Capítulo I.

II.4.1 El tipo enumerado

Crear una enumeración es definir un nuevo tipo de datos, denominado "tipo enumerado" y declarar una variable de este tipo. La sintaxis es la siguiente:

```
enum tipo_enumerado{
  <definición de nombres de constantes enteras>
};
```

donde tipo_enumerado es un identificador que nombra al nuevo tipo definido.

El siguiente ejemplo declara una variable llamada *color* del tipo enumerado colores, la cual puede tomar cualquier valor especificado en la lista.

```
enum  colores{azul, amarillo, rojo, verde, blanco, negro};

colores color;
color = azul;
```

Cada identificador de la lista de constantes en una enumeración, tiene asociado un valor. Por default, el primer identificador tiene asociado el valor "0", el siguiente el valor "1" y así sucesivamente, de tal forma que:

```
color = verde;      es equivalente a      color = 3;
```

En C++ un valor de tipo `int` no puede ser asignado directamente a un tipo enumerado, por lo que es necesario hacer una "conversión explícita del tipo" de la siguiente forma:

```
color = (colores)3;
```

Los tipos enumerados ayudan a acercar mas el lenguaje de alto nivel a nuestra forma de expresarnos, así la expresión: "si el color es verde,..." dice mas que la expresión "si el color es 3,…"

Podemos definir un *tipo de estudiante* de la siguiente forma:

```
enum tipo_estudiante{ flojo, atarantado, estudioso, nerd };
```

Declaramos la variable *`seleccion`* para preguntar por el tipo de estudiante:

```
int seleccion;
```

Luego usamos un menú para preguntar al usuario el tipo de estudiante que considera ser:

```
cout << "¿Como estudias para un examen?";
cout << endl << " elige uno de los siguientes numeros: \n"
     <<" 0 : no estudio"<<endl
     <<" 1 : estudio un dia antes"<<endl
     <<" 2 : repaso cada clase y pregunto mis dudas"<<endl
     <<" 3 : estudio a todas horas todos los dias"<<endl;

cin >> seleccion;
```

Ahora declaramos una variable `alumno` de tipo `tipo_estudiante`:

```
tipo_estudiante alumno;
```

Si quisiéramos asignar el valor capturado en la variable `selección` en la variable `alumno`, entonces:

```
int seleccion;
alumno = (tipo_estudiante)seleccion;
```

Las siguientes asignaciones son equivalentes:

```
  alumno = (tipo_estudiante)3;
  alumno = nerd;
```

II.4.2 Validación de la entrada con *do-while*

Para validar que la entrada que proporciona el usuario esté dentro del rango que se espera se puede utilizar la estructura *do-while*. Por ejemplo, para validar que la entrada del menú de la sección anterior anterior sea 0, 1, 2 ó 3 hacemos lo siguiente:

```
int seleccion;
cout << "¿Como estudias para un examen?";
do {
  cout << endl << " elige uno de los siguientes numeros: \n"
       <<" 0 : no estudio"<<endl
       <<" 1 : estudio un dia antes"<<endl
       <<" 2 : repaso cada clase y pregunto mis dudas"<<endl
       <<" 3 : estudio a todas horas todos los dias"<<endl;

  cin >> seleccion;
}while( ( seleccion < flojo )||( selección > nerd ));
```

El ciclo se ejecuta mientras que el usuario proporcione un dato equivocado

De esta forma protegemos al programa de los que no entienden las instrucciones. Asi, cuando el usuario proporciona un dato correcto el ciclo ya no se ejecuta y el programa continúa con la siguiente instrucción después del *do-while*.

Ejercicio 2.4.1.- Validar que el usuario proporcione un número entre 0 y 50.

Solución: En la Figura II-58 se observa el diagrama de flujo en la que se emplea la estructura *do-while*. Nótese que la condición debe **ser verdadera** cuando la salida esta **equivocada**, , es decir cuando x<0 OR x >50, solo de esta forma se repite el ciclo una y otra vez, solo se detiene hasta que el usuario introduzca el dato correctamente.

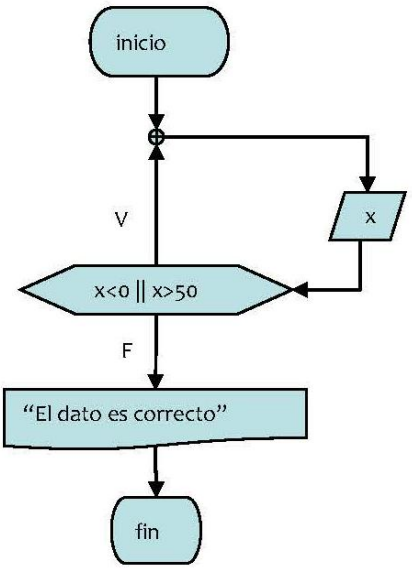


Figura II-58 Validando que el dato proporcionado esté entre el 0 y el 50

El código correspondiente al diagrama de flujo de la Figura II-58 es el siguiente.

```
// obtiene un número entre 0 y 50
#include <iostream.h>

main() {
  int x;

  cout << "Da un numero entre 0 y 50";
  do{
    cin >> x;
  }while( x < 0 || x > 50 );

  cout << "Lo lograste! El dato es: " << x;

  return 0;
}
```

Ejercicio 2.4.2.- Validar que el usuario proporcione un número entre -25 y 32.

Solución: En la Figura II-59 se observa el diagrama de flujo en la que se emplea la estructura *do-while.* Nótese que la condición debe **ser verdadera** cuando la salida esta **equivocada**, es decir cuando x<-25 OR x >32, solo de esta forma se repite el ciclo una y otra vez, solo se detiene hasta que el usuario introduzca el dato correctamente.

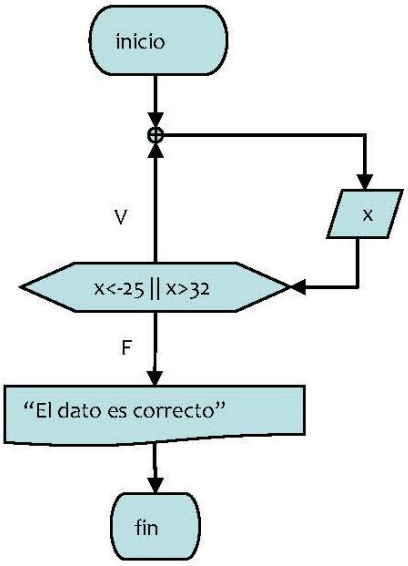


Figura II-59 *Validando que el dato proporcionado esté entre -25 y 32*

El código correspondiente al diagrama de flujo de la Figura II-59 es el siguiente.

```
// obtiene un número entre -25 y 32
#include <iostream.h>

main() {
  int x;

  cout << "Da un numero entre -25 y 32";
  do{
    cin >> x;
  }while( x < -25 || x > 32 );

  cout << "Lo lograste! El dato es: " << x;

  return 0;
}
```

II.4.3 Anidación de estructuras repetitivas

Así como podemos anidar varios *if* también es posible anidar el *while* y el *do-while*. Por ejemplo, podemos usar un *do-while* para que se ejecute un menú un número indeterminado de veces, hasta que el usuario oprima ESC y dentro de este ciclo, anidar otro *do-while* para validar el dato de entrada. Como se indica en la Figura II-60.

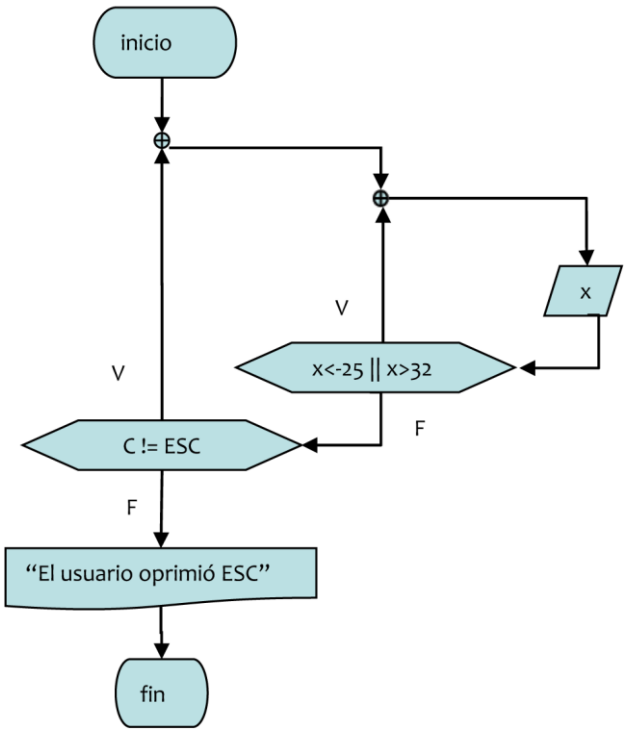


Figura II-60 *Ejecución de un do-while hasta que el usuario oprima la tecla ESC*

El código correspondiente al diagrama de flujo de la Figura II-60 es el siguiente. La instrucción `getch()` sirve para capturar un carácter del teclado y se encuentra en `conio.h`.

```
// obtiene un número entre -25 y 32
#include <iostream.h>
#include <conio.h>

main() {
  int x;
  char c, cr[2];

  do{
    do{
      cout << "Da un numero entre -25 y 32 \n";
      cin >> x;
    }while( x < -25 || x > 32 );

    cout << "Lo lograste! El dato es: " << x;
    cout << "\n oprime ""ESC"" para terminar";
    c = getch();
    cin.getline(cr,2);

  }while(27 != c);

  cout << "\n El usuario oprimio ESC…Adios!!";

  return 0;
}
```

Ojo! Esto es nuevo!

El problemita del `getch()`:

Como C++ es un lenguaje diseñado principalmente para trabajar con números, existen algunos detalles que hay que tomar en cuenta al trabajar con cadenas de caracteres. Despues de utilizar un `getch()` debemos usar un `cin.getline( )`, para retirar "la basura" que queda en el buffer de captura de datos, asi:

```
char c, cr[2];

c = getch();

cin.getline( cr, 2 );
```

Con getch se toma el dato pero el ENTER se queda en el buffer

getline quita el ENTER del buffer

A continuación presentamos el código de un programa que combina la estructura repetitiva *do-while* anidada con la estructura selectiva múltiple para elaborar un menú.

```
// Ejemplo de menú con enum y un ciclo do / while
#include<iostream.h>
#include<conio.h>

main() {
  char c, cr[2];
  enum tipo_estudiante{ flojo, atarantado, estudioso, nerd};
  int seleccion;

  do {
    cout << "¿Como estudias para un examen?";
    do {
      cout << endl << "elige una de las siguientes opciones: \n"
           <<" 0 : no estudio"<<endl
           <<" 1 : estudio un dia antes"<<endl
           <<" 2 : repaso cada clase y pregunto mis dudas"<<endl
           <<" 3 : estudio a todas horas todos los dias"<<endl;
      cin >> seleccion;
    }while(( seleccion < flojo ) || ( seleccion > nerd ));

    switch(seleccion) {

      case flojo:
        cout<<" Mejor maneja un taxi! \n";
        break;

      case atarantado:
        cout<<"Piensa que el examen ya es mañana\n";
        break;

      case estudioso:
        cout<<"Estudia ingles y elige a que país vas a pasear \n";
        break;

      case nerd:
        cout<<"tambien hay personas, arte, deportes, etc. \n"
            <<"Vive!!";
        break;
    };

    cout<<"oprime ""Esc"" para terminar"<<endl;
    c = getch();
    cin.getline(cr,2);
  }while(27 != c);

  cout << " Adios! ";
  return 0;
}
```

Para que aparezca entre comillas

Lo que esta en este ciclo do-while se ejecuta hasta que el usuario oprima la tecla ESCAPE (su código es 27)

Capítulo III Estructuras de datos

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

Comprender los conceptos y elaborar programas en C++ que contengan:

- Arreglos
- Registros
- Combinaciones entre arreglos y registros

III.1 Arreglos

III.1.1 Definición de arreglo

> "Un arreglo es un conjunto *finito* y *ordenado* de elementos homogéneos."

- *Finito*: porque todo arreglo tiene un límite, es decir, debe determinarse cual será el número máximo *n* de elementos que podrán formar parte del arreglo.
- *Ordenado*: porque cada elemento se localiza por medio de un índice que va de 0 a *n*-1.
- *Homogéneo*: porque todos los elementos de un arreglo son del mismo tipo (todos enteros, todos de punto flotante, etc.), pero nunca combinaciones entre distintos tipos.

Para declarar un arrego en C y en C++ se escribe primero el tipo de dato de sus elementos, luego se da el nombre del arreglo y finalmente el número de elementos que contendrá. La sintaxis es la siguiente:

```
<tipo>   <nombre> [ <tamaño> ];
```

El *nombre* del arreglo sirve para hacer referencia a él en el programa. Para hacer referencia a alguno de sus elementos se usa, además del nombre, un *índice* (entre corchetes) que es un número entero correspondiente al lugar que ocupa cada elemento en el arreglo.

Por ejemplo, un arreglo de *enteros* llamado *edades* con *8* elementos se declara como:

```
int   edades [ 8 ];
```

Así el arreglo `edades` tiene las partes que se indican en la Figura III-1. En este caso `n` =8, es decir, el arreglo `edades` es de tamaño 8 y sus ìndices van de 0 a 7.

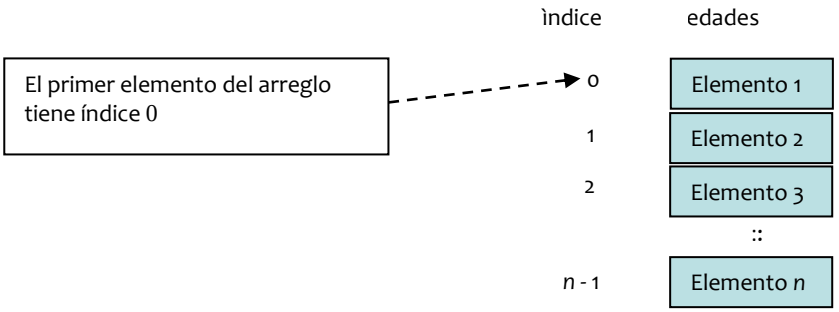


Figura III-1 *Partes del arreglo edades.*

Otro ejemplo, un arreglo de *float* llamado *calificaciones* con *5* elementos:

```
float   calificaciones [ 5 ];
```

El arreglo `calificaciones` tiene las partes que se indican en la Figura III-2

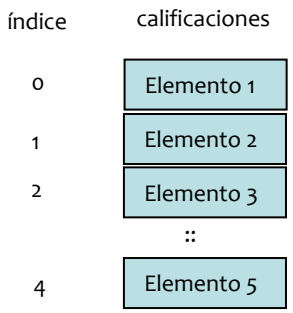


Figura III-2 *Partes del arreglo `calificaciones`.*

Para acceder al *contenido de un elemento de un arreglo* se utiliza el nombre del arreglo y se indica entre corchetes cuadrados "`[ ]`" el índice del arreglo donde se encuentra el elemento deseado.

Ejemplo 3.1.- si el contenido del arreglo `edades` es el que se muestra en la Figura III-3:

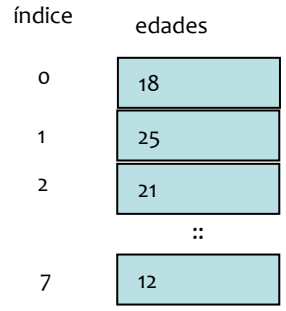


Figura III-3 *Contenido del arreglo* `edades`.

Cual sería el contenido de las siguientes variables?

```
a = edades[ 0 ];                a = ?
b = edades[ 2 ];                b = ?
```

Respuesta:

```
a = 18
b = 21
```

Cual sería el contenido de las siguientes variables?

```
c = edades[ 7 ];                c = ?
d = edades[ 8 ];                d = ?
```

Respuesta:

```
c = 12
d = ?
```

En el último caso no es posible asegurar cuál será el valor de `d` ya que `edades[ 8 ]` no forma parte del arreglo. El compilador de C++ no produce ningún error si se intenta acceder a un arreglo con un valor de índice fuera del rango definido. El problema se presentará durante la ejecución del programa, pues se trabaja con datos desconocidos.

Error común: Es importante hacer notar la diferencia entre el "*i*-ésimo elemento del arreglo" y el "elemento con subíndice *i* en el arreglo".

El i-ésimo elemento de un arreglo tiene subíndice "i-1".

Por ejemplo: el tercer elemento del arreglo `edades` de la Figura III-3 es:

```
edades[2]
```

El octavo elemento del arreglo `edades` de la Figura III-3 es:

```
edades[7]
```

Todo lo anterior es porque los arreglos en C y C++ empiezan con el índice CERO a diferencia de otros lenguajes en los que el primer elemento tiene índice 1

Ejemplos 3.2.- declarar:

a).- Un arreglo llamado "Enteros" que almacenará 10 enteros

```
int Enteros[10];
```

b).- Un arreglo llamado "Reales" que almacenará 5 Valores de tipo `double`

```
double Reales[5];
```

c).- Un arreglo llamado "Caracteres" que almacenará 11 caracteres

```
char  Caracteres[11];
```

d).- Un arreglo llamado "Clase" que almacenará calificaciones de 25 estudiantes de tipo `float`.

```
float  Clase[25];
```

III.1.2 Forma de trabajar con los arreglos

Para recorrer un arreglo en C++ utilizamos un ciclo `for`.

Ejemplo 3.3.- pedir al usuario el valor de los de un arreglo de enteros:

```
// Recorrido de arreglos ArregloRecorre.cpp
#include<iostream.h>

main() {
  int edades[8];

  for( int i = 0; i < 8; i++ ){
    cout << " cual es la edad numero: " << i+1 << "?";
    cin >> edades[ i ];
  };

  cout << " ya tengo la informacion! " << endl
       << " pero no la pongo en la pantalla ";

  return 0;
}
```

Para desplegar el contenido del arreglo, **no basta** con poner `cout << edades[i]`, ya que la `i` es un índice que va de `0` a `n-1`, luego entonces, es necesario usar siempre el ciclo `for` cuando se trabaja con arreglos.

Las tres operaciones básicas con los arreglos son:

Pedir los datos de un arreglo.
Recorrer un arreglo para trabajar con sus datos.
Desplegar en pantalla los datos del arreglo.

Así que para mostrar en pantalla los datos del arreglo `edades` del ejemplo 3.3 tendremos:

```
// Recorrido de arreglos ArregloRecorre.cpp
#include<iostream.h>

main() {
  int edades[8];
  for( int i = 0; i < 8; i++ ){
    cout << " cual es la edad numero " << i+1 << "?";
    cin >> edades[ i ];
  };
  for( int i = 0; i < 8; i++ ){
    cout << " edades[ " << i << "]= "
         << edades[ i ] << endl;
  };

  return 0;
}
```

Puntos a tomar en cuenta cuando se trabaja con arreglos:

Cuando recorremos un arreglo utilizando un ciclo, el índice nunca debe ser menor que "cero".

El índice siempre deberá ser menor que el número total de elementos que tenga el arreglo (uno menos que el tamaño del mismo).

Asegurarse de que la condición de terminación del ciclo **no** permita que se haga referencia a un elemento que esté **fuera de los límites** del arreglo, ya que éste es un error de lógica en tiempo de ejecución, no es un error de sintaxis. Por ejemplo, si tenemos declarado el arreglo `A` como `int A[ 5 ];`, nunca se debe acceder a `A[ 5 ]`.

Pidiendo datos de los arreglos.

A continuación se muestran dos ejemplos de cómo pedir los datos de un arreglo

a).- Para el arreglo de enteros:

int Enteros[10];

Como ya se mencionó anteriormente, debemos usar el ciclo `for` para trabajar con los arreglos:

```
for( int i = 0; i < 10; i++ ){
  cout << "\n Enteros[ " << i << " ]=?";
  cin >> Enteros[ i ];
};
```

Para visualizar qué pasa con cada iteración del ciclo `for`, incluimos la prueba de escritorio en la Figura III-4:

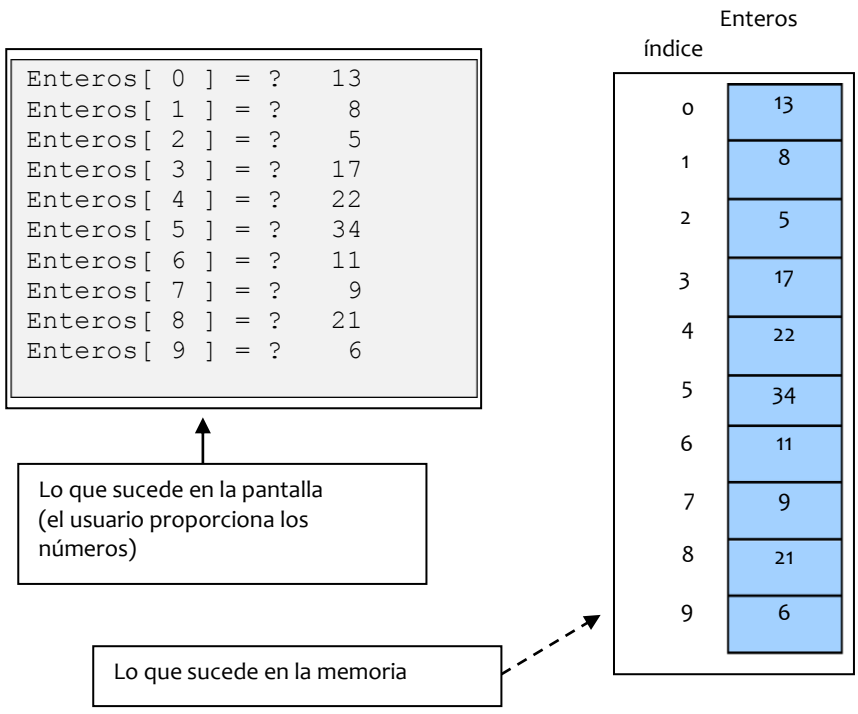


Figura III-4 *Prueba de escritorio del programa que pide los datos del arreglo `enteros[10]`.*

b).- Un arreglo llamado "Reales" que almacenará 5 valores de tipo `double`

```
double Reales[5];
for( int i = 0; i < 5; i++ ){
  cout << "\n Reales[ " << i << " ]=?";
  cin >> Reales[ i ];
};
```

Para visualizar qué pasa con cada iteración del ciclo `for`, incluimos la prueba de escritorio en la Figura III-5.

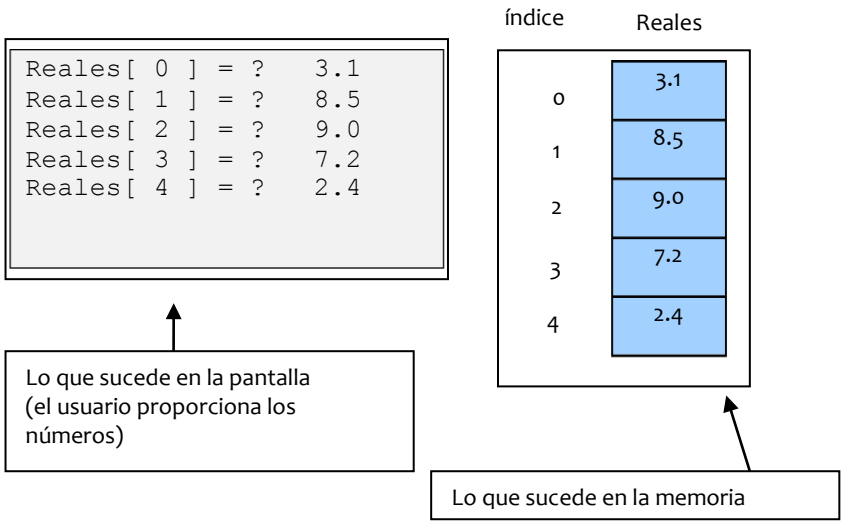


Figura III-5 *Prueba de escritorio del programa que pide los datos del arreglo* `reales[5]` .

III.1.3 Ejercicios con arreglos

Ejercicio 3.1.- Pedir los datos del arreglo: `float Clase[25];`

Solución:

```
for( int i = 0; i < 25; i++ ){
  cout << "\n Clase[ " << i << " ]=?";
  cin >> Clase[ i ];
};
```

Ejercicio 3.2.- Pedir los datos del arreglo: `double Datos[300];`

Solución:

```
for( int i = 0; i < 300; i++ ){
  cout << "\n Datos[ " << i << " ]=?";
  cin >> Datos[ i ];
};
```

Ejercicio 3.3.- Pedir los datos del arreglo: `int Precios[5000];`

Solución:

```
for( int i = 0; i < 5000; i++ ){
  cout << "\n Precios[ " << i << " ]=?";
  cin >> Precios[ i ];
};
```

Ejercicio 3.4.- Hacer un programa que encuentre el elemento mayor de un arreglo `A` de enteros de tamaño 8, con los siguientes elementos: `{ 4, 3, 21, 1, 25, 4, 10 }`.

```
// mayor.cpp
// Programa que encuentra el mayor de 8 elementos
#include <iostream.h>

main() {
  int A[ 7 ] = { 4, 3, 21, 1, 25, 4, 10 };
  int mayor = A[ 0 ];
  for( int i = 1; i < 7; i++ ){
   if( A[ i ] > mayor ) {
      mayor = A[ i ];
    };
  };

  cout<< "el elemento mayor del arreglo A es: " << mayor;
  return 0;
}
```

valores iniciales del arreglo

Tomamos como referencia el primer elemento del arreglo

En la Figura III-6 mostramos la prueba de escritorio:

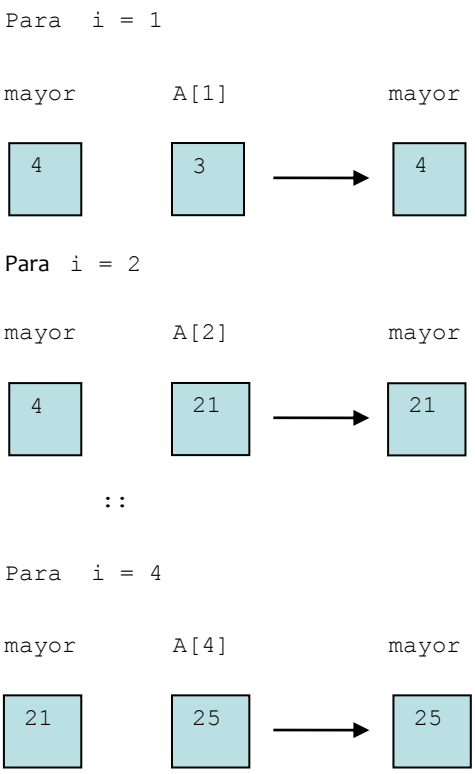


Figura III-6 *Prueba de escritorio de la búsqueda del elemento mayor en `A[7]`*

Para los casos i =5, i =6 no se cumple la condición: `A[ i ]> mayor`, por lo que no se altera el valor de la variable `mayor`.

Ejercicio 3.5.- Hacer un programa que saque el promedio de 5 calificaciones contenidas en un arreglo. Primero hay que pedir las calificaciones y guardarlas.

```
//programa que saca el promedio de 5 calificaciones promedio.cpp
#include <iostream.h>

main() {
  float calif[5], suma=0, promedio;

  // pedir los datos
  for( int j = 0; j < 5; j++ ){
    cout << "introduce el elemento" >> j+1;
    cin >> calif[ j ];
  };

  //obtener el promedio
  for( int i = 0; i < 5; i++ )
    suma = suma + calif[ i ];

  promedio = suma / 5;
  cout << "el promedio es " << promedio << endl;
  return 0;
}
```

En la Figura III-7 mostramos la prueba de escritorio:

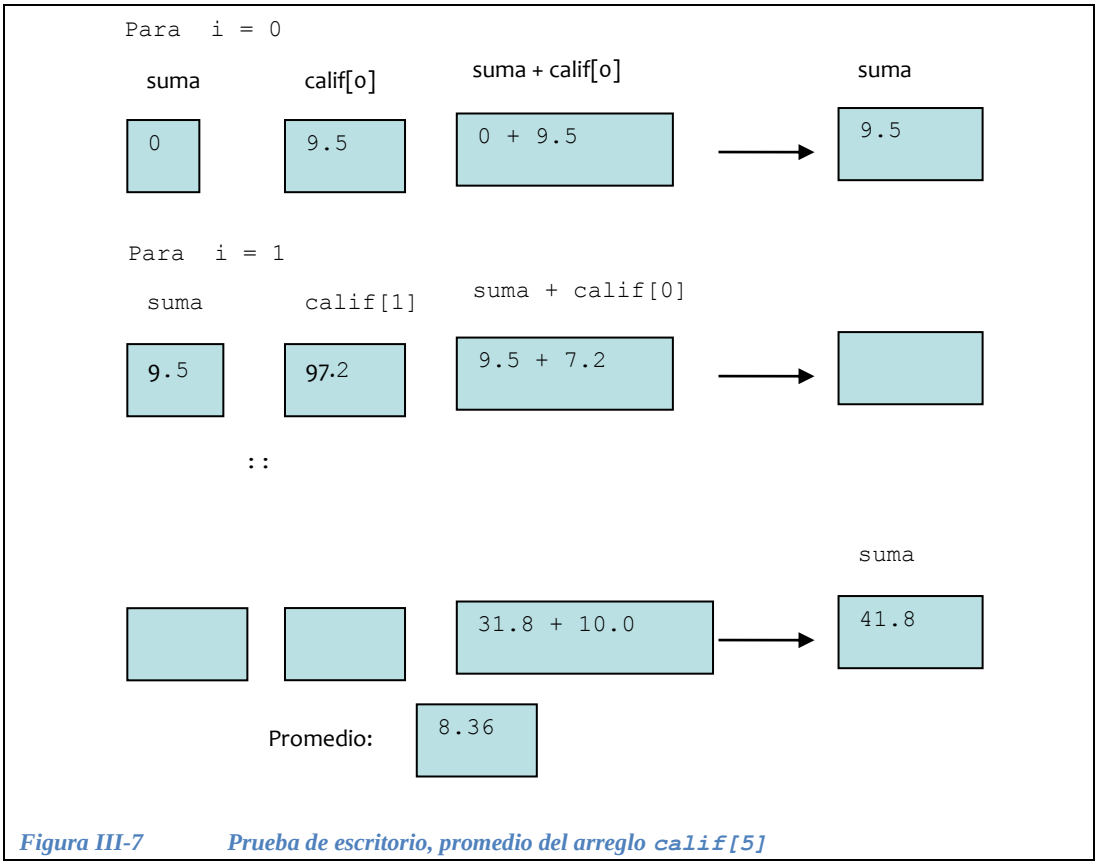


Figura III-7 *Prueba de escritorio, promedio del arreglo* `calif[5]`

III.2 Registros

III.2.1 Definición de registro

> "Los registros son estructuras que permiten almacenar varios datos (de tipos distintos o no) bajo un mismo nombre."

Un registro es una estructura de datos en la que se agrupan y almacenan varios datos. A estos datos se les llama campos del registro, los campos no tienen que ser del mismo tipo necesariamente. Cada uno de los campos se identifica con un nombre. Al registro se le da también un nombre. El nombre del registro se usa como un tipo de dato definido por el programador, es posible declarar variables cuyo tipo es un registro. Las variables de tipo registro almacenan cada uno de los campos del registro.

III.2.2 Forma de trabajar con los registros.

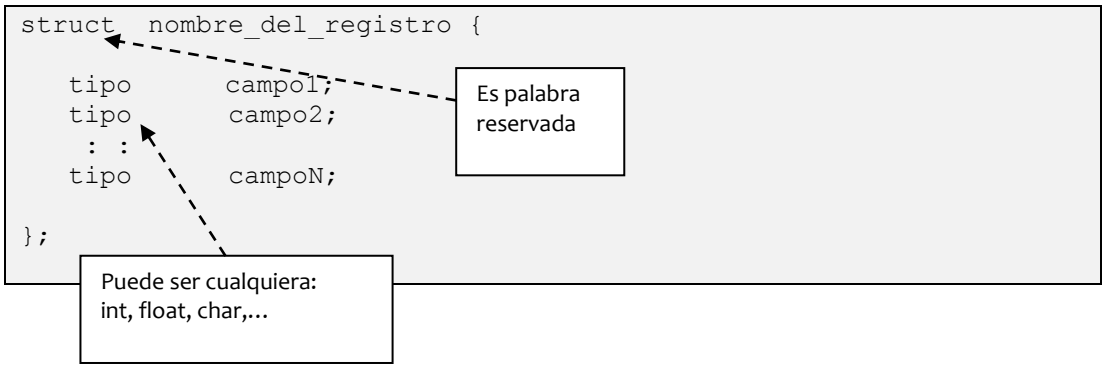
Para declarar un registro y su nombre en C++, se escribe:

```
struct <nombreReg> {
  <lista de declaraciones>
};
```

<nombreReg> es el identificador del registro.

<lista de declaraciones> es en donde se declara el tipo y el nombre de cada uno de los campos en el registro.

Así:



Para declarar una variable de tipo registro se usa el nombre del registro de la misma forma en que se usan los tipos de dato existentes.

```
<nombreReg> <nombreVar>;
```

La variable <nombreVar> es de tipo <nombreReg>.

Ejemplo 3.3.- Declaración del *registro* s_tiempo:

```
struct s_tiempo {
   int hora;
   int minutos;
   int segundos;
};
```

Declaración de variables de tipo s_tiempo:

Podemos tener cualquier cantidad de variables del nuevo tipo: s_tiempo

```
s_tiempo   tiempo_del_proceso;
s_tiempo   tiempo_de_captura;
           ::
s_tiempo   tiempo_total;
```

Podemos tener cualquier cantidad de variables del nuevo tipo: s_tiempo

Ejemplo 3.4.- Declaración del *registro* `s_direccion`:

```
struct   s_direccion{
   char   calle[20];
   int    numero;
   char   colonia[30];
   int    codigo_postal;
   char   ciudad[15];
};
```

Una vez definido el tipo agregado `s_direccion` se procede a declarar las variables necesarias:

```
s_direccion   domicilioCarlos;
s_direccion   domicilioBeto;
```

Ejemplo 3.5.-

a).- Declarar un registro llamado s_coordenadas que tenga dos campos de tipo entero llamados `x` y `y`.

Solución:

```
struct s_coordenadas {
  int x;
  int y;
};
```

ó también:

```
struct s_coordenadas {
  int x, y;
};
```

b).- Declarar una variable de tipo `s_coordenadas` llamada `posicion`.

Solución:

```
s_coordenadas posicion;
```

c).- Declarar una Variable de tipo arreglo de 4 `s_coordenadas` llamado `cuadrilatero`

Solución:

```
s_coordenadas cuadrilatero[4];
```

Acceso a los registros.

Para tener acceso a los campos de una variable de tipo registro, es necesario anteponer el nombre de la variable seguida de un punto y sin dejar espacio entre el punto y el nombre del campo.

Ejemplo 3.6.- para el registro:

```
struct s_tiempo {
  int hora;
  int minutos;
  int segundos;
};

s_tiempo   inicio;
s_tiempo   fin;

inicio.hora = 7;
inicio.minutos = 26;
inicio.segundos = 5;

fin.hora = 8;
fin.minutos = 30;
fin.segundos = 12;
```

El contenido de los registros puede apreciarse en la Figura III-8.

Figura III-8 *Contenido de los registros `inicio` y `fin`.*

Ejemplo 3.7.- acceso a registros poniendo los datos desde el programa:

```
struct   s_direccion{
   char     calle[20];
   int      numero;
   char     colonia[30];
   int      codigo_postal;
   char     ciudad[15];
};

s_direccion   domicilioMario;
s_direccion   domicilioBeto;

strcpy(domicilioMario.calle, "de la amargura");
domicilioMario.numero = 1000;
strcpy(domicilioMario.colonia, "pedreras");
domicilioMario.codigo_postal = 03045;
strcpy(domicilioMario.ciudad, "Mexico");
```

Nota: la función `strcpy` se usa para copiar datos (palabras) a una cadena de caracteres desde el programa, su sintaxis es la siguiente:

```
strcpy( cadena_destino, "datos a guardar en la cadena");
```

En la Figura III-9 se muestra como queda en la memoria el contenido del registro `domicilioMario`

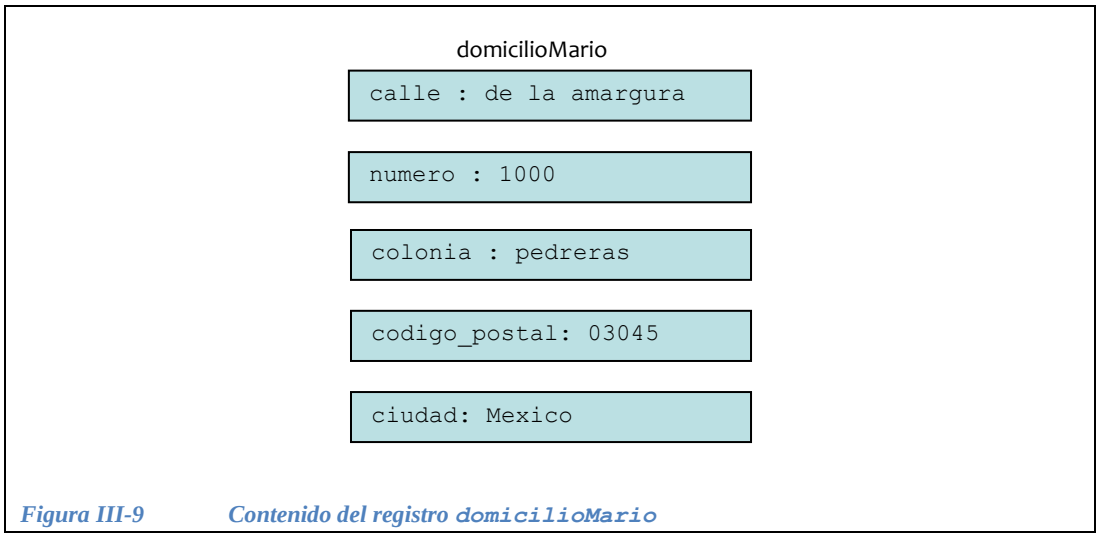


Figura III-9 *Contenido del registro `domicilioMario`*

Ejemplo 3.8.- acceso a registros pidiendo los datos al usuario:

```
cout << " Calle? ";
cin.getline(domicilioBeto.calle, 20);
cout << " numero? ";
cin >> domicilioBeto.numero;
cin.getline(cr,2);
cout << " Colonia? ";
cin.getline(domicilioBeto.colonia, 30);
cout << " codigo postal? ";
cin >> domicilioBeto.codigo_postal;
cin.getline(cr,2);
cout << " Ciudad? ";
cin.getline(domicilioBeto.ciudad, 15);
```

Nota: la función `cin.getline` se usa para la entrada de arreglos de caracteres, su sintaxis es la siguiente:

cin.getline(cadena_destino, numero máximo de caracteres a capturar);

En la Figura III-10 se muestra como queda en la memoria el contenido del registro `domicilioBeto`

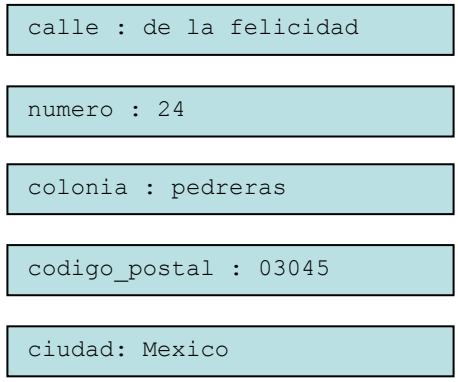


Figura III-10 *Contenido del registro* `domicilioBeto`

Nótese que en este caso los datos los proporcionó el usuario!

III.2.3 Ejercicios con registros

Ejercicio 3.6.- Declarar un registro llamado `s_inscripcion` que tenga los campos: nombre (cadena de 20 caracteres), edad (entero), peso (`float`), estatura (`float`), sexo (`char`), grado(`int`).

Solución:

```
struct s_inscripcion {
  char  nombre[20];
  int   edad;
  float peso, estatura;
  char  sexo;
  int   grado;
};
```

Ejercicio 3.7.-

a).- Declarar un registro llamado `s_transaccion` con los campos: opcion (entero), cuenta (`double`), monto (`double`), anio (`int`), mes (`int`), dia (`int`), hora (`int`), minuto (`int`), segundo (`int`).

Solución:

```
struct s_transaccion {
  int    opcion;
  double cuenta;
  double monto;
  int    anio, mes, dia;
  int    hora, minuto, segundo;
};
```

b).- Declarar la variable tr de tipo s_transaccion

Solución:

s_transaccion tr;

c).- ¿Como se tiene acceso al campo cantidad?

Solución:

tr.cantidad;

d).- ¿Como se asigna el valor 30.5 al campo cantidad?

Solución:

tr.cantidad = 30.5;

El Valor que contiene este campo se usa siempre anteponiendo la Variable de tipo registro seguida del punto, por ejemplo:

cout << tr.cantidad;

Ejercicio 3.8.- Si se tiene el registro `s_info_personal`:

```
struct s_info_personal {
  int   edad;
  char  sexo;
  char  nombre[40];
  float peso, estatura;
};
```

y la variable `portero` de tipo `s_info_personal` hacer un programa escriba en pantalla el valor leído en cada campo de la variable `portero`.

Solución:

```
#include <iostream.h>

struct s_info_personal {
  int   edad;
  char  sexo;
  char  nombre[40];
  float peso, estatura;
};

main() {
  s_info_personal portero;
  char cr[2];
  cout << "edad:"; cin >> portero.edad;
  cout << "sexo:"; cin >> portero.sexo;
  cin.getline(cr,2);
  cout << "nombre:"; cin >> portero.nombre;
  cout << "peso:"; cin >> portero.peso;
  cout << "estatura:"; cin >> portero.estatura;

  cout << "edad:" << portero.edad << endl
       << "sexo:" << portero.sexo << endl
       << "nombre:" << portero.nombre << endl
       << "peso:" << portero.peso << endl
       << "estatura:" << portero.estatura << endl;

  return 0;
}
```

No necesitamos cin.getline cuando usamos una sola palabra (no hay espacios).

Comparación de arreglos con registros.

Arreglos	Registros
- Almacena N elementos del mismo tipo.	- Almacena N elementos de diferentes tipos.
- Sus componentes se accesan por medio de índices	- Sus componentes se llaman *campos* y se accesan por medio de su nombre.
- El orden entre sus elementos se establece por su índice.	- El orden entre los campos no es importante

Tabla 7 Comparación de arreglos con registros.

III.3 Combinaciones entre arreglos y registros

Los arreglos se pueden combinar con los registros para formar estructuras más complejas, esto da mayor flexibilidad a la programación estructurada. En la Figura III-11 se muestran las combinaciones más utilizadas:

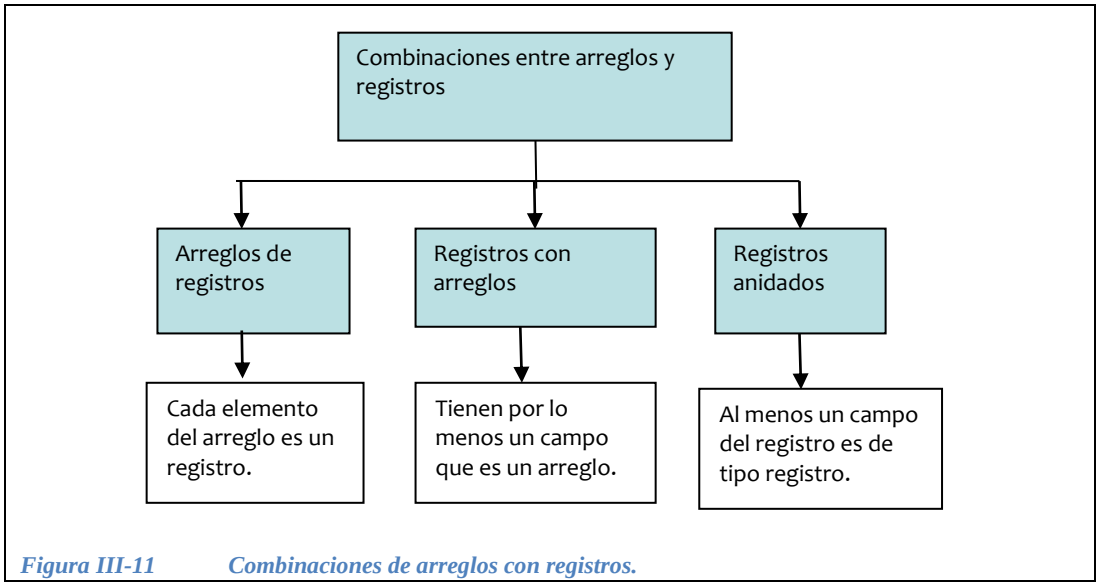


Figura III-11 Combinaciones de arreglos con registros.

III.3.1 Arreglos de registros.

En este tipo de combinación cada elemento del arreglo es un registro, esto hace posible agrupar `n` registros en una sola estructura, por ejemplo, si se tiene el registro `s_calific:`

```
struct  s_calific{
  float  calificacion;
  long   materia;
};
```

Ejemplo 3.8.- Podemos declarar un arreglo llamado `calificaciones` que contenga seis registros `s_calific` de la siguiente forma:

s_calific calificaciones[6];

De tal forma que e en la memoria tendríamos algo similar al diagrama de la Figura III-12:

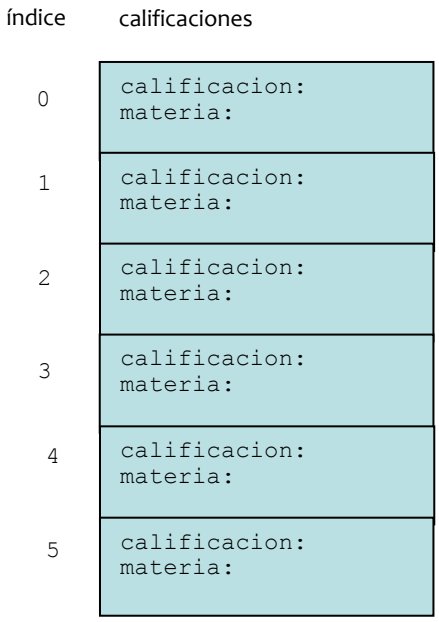


Figura III-12 *Arreglo de seis registros de tipo* `s_califi c.`

¿Cómo se piden los datos para el arreglo `calificaciones` ?

Solución:

```
for( int i=0; i<6 ; i++ ){
  cout << "calificacion " << i+1 << " =?";
  cin >> calificaciones[i].calificacion;
  cout << "materia " << i+1 << " =?";
  cin >> calificaciones[i].materia;
};
```

Seguimos utilizando el índice para referirnos a cada elemento del arreglo

Seguimos utilizando el punto para referirnos a cada campo del registro

En la Figura III-13 se puede ver la prueba de escritorio, del lado izquierdo lo que saldría en pantalla con datos que introduce el usuario. Y del lado derecho, el mapa del arreglo de registros en memoria, ya con los datos proporcionados por el usuario.

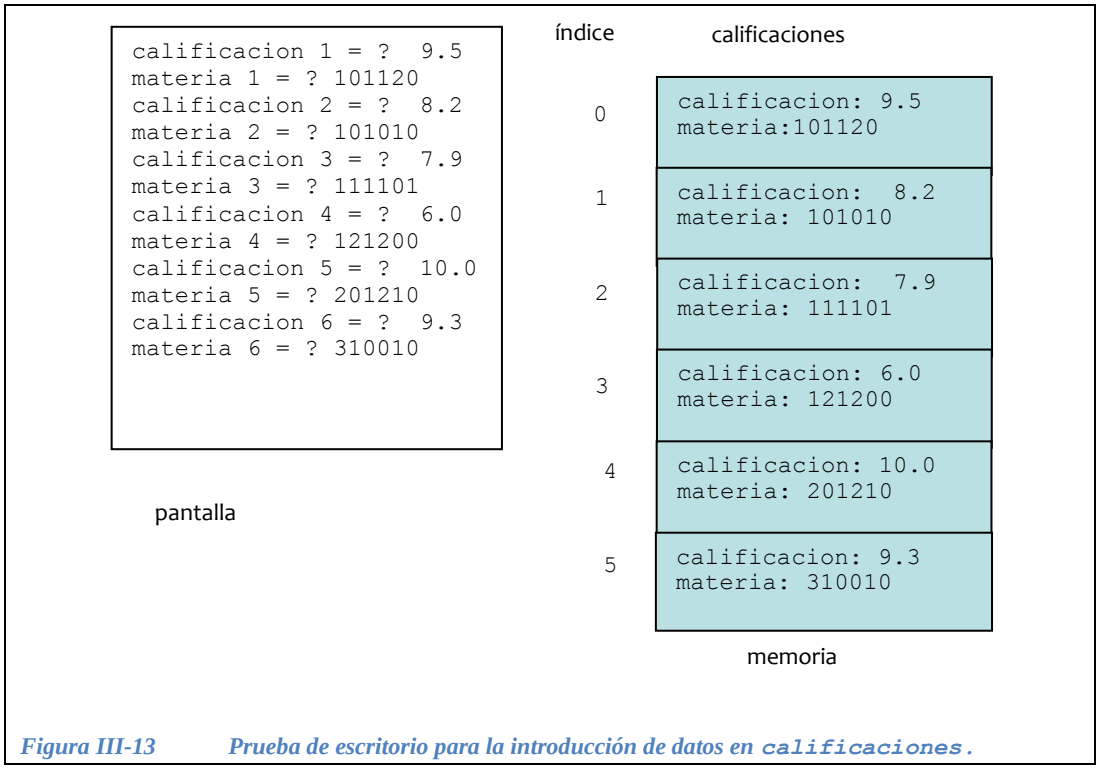


Figura III-13 *Prueba de escritorio para la introducción de datos en* `calificaciones`.

III.3.1.1 Ejercicos con arreglos de registros

Ejercicio 3.9.- Haz un programa que pida los datos del arreglo de registros anterior y que obtenga el promedio de la calificaciones de todas las materias.

Solución:

```
#include<iostream.h>

struct s_calific{
  long   materia;
  float  calificacion;
};

s_calific  calificaciones[6];
char cr[2];
int i;
float suma=0;

main() {
  // pedir los datos
  for( i=0; i<6; i++ ){
    cout << "calificación? " << i+1 << " =? ";
    cin >> calificaciones[i].c_calificacion;
    cout << "materia" << i+1 <<" =? ";
    cin >> calificaciones[i].materia;
  };
  // obtener el promedio
  for( i=0; i<6; i++ )
    suma = suma + calificaciones[i].calificacion;

  cout<< " tu promedio es" << suma/6;
  return 0;
}
```

Ejercicio 3.10.- Hacer un programa que capture la información detallada de 100 transacciones bancarias y que la escriba en un archivo de disco. El archivo deberá tener en cada renglón la información de una y solamente una transacción. Cada transacción es del tipo `s_transaccion` del ejercicio 3 de la seeción III.2:

Los campos que deben capturarse para cada transacción son

- Fecha (año, mes, día)
- Hora (hora, minuto, segundo)
- Tipo de transacción (un entero entre 1 y 2)
- Número de cuenta (de 10 dígitos)
- Monto de la transacción (con punto decimal)

Solución:

```
#include <iostream.h>
#include <fstream.h>

struct s_transaccion {
  int    anio, mes, dia;
  int    hora, minuto, segundo;
  int    opcion;
  double cuenta;
  double monto;
};

#define NUM_TR 100

main() {
  // declaración del arreglo de regristros de tamaño 100
  s_transaccion tr[NUM_TR];

  // Solicitar los datos de los 100 registros al usuario
  for( int i=0; i<NUM_TR; i++ ) {
    cout << "Fecha de la transacción? (año mes día)";
    cin >> tr[i].anio >> tr[i].mes
        >> tr[i].dia;
    cout << "Hora de la transacción? (hora minuto segundo)";
    cin >> tr[i].hora >> tr.minuto
        >> tr[i].segundo;
    cout << "opción? ( 1: depósito, 2: retiro )";
    cin >> tr[i].opcion;
    cout << "numero de cuenta? (10 dígitos)";
    cin >> tr[i].cuenta;
    cout << "monto de la transacción?
    cin >> tr[i].monto;
  };

  // abrir el archivo de salida
  ofstream salida;
  salida.open("transacciones.dat")

  // Escribe el encabezado del archivo
  cout << " año mes día hora minuto segundo opción cuenta monto";

  // Escribe los datos por renglón para cada uno de los registros
  for( int i=0; i<NUM_TR; i++ ) {
    salida << tr[i].anio
      << tr[i].mes
      << tr[i].dia
      << tr[i].hora
      << tr[i].minuto
      << tr[i].segundo
      << tr[i].opcion
      << tr[i].cuenta
      << tr[i].monto << endl;
  };
  salida.close();
  return 0
}
```

Definimos que la constante `NUM_TR` contenga el número `100`

Ejercicio 3.11.- Hacer un programa que lea la información del archivo generado con el programa anterior y que escriba en pantalla el total de los montos en las transacciones así como el promedio de estos montos.

Solución:

```
#include <iostream.h>
#include <fstream.h>

struct s_transaccion {
  int anio, mes, dia;
  int hora, minuto, segundo;
  int opcion;
  double cuenta;
  double monto;
};

#define NUM_TR 100

main() {
  s_transaccion tr; // Variable de tipo registro en donde se van a
                    // leer los datos del archivo

  char cr[50]; // arreglo de caracteres para guardar el encabezado

  ifstream ent;
  ent.open("transacciones.dat");
  int i;
  double suma = 0;
  double promedio;

  // primero leer el encabezado del archivo
  cin.getline(cr,50);

  // leer los datos del archivo
  for( i=0; i<NUM_TR; i++ ) {
      ent >> tr.anio >> tr.mes
            >> tr.dia;
      ent >> tr.hora >> tr.minuto
            >> tr.segundo;
      ent >> tr.opcion;
      ent >> tr.cuenta;
      ent >> tr.monto;
      suma = suma + tr.monto;
    } ;

    // cerrar el archivo cuando se termina de leer
    ent.close();

    promedio = suma / NUM_TR;
    cout << "suma=" << suma << endl
            << "prom=" << promedio;
    return 0;
}
```

III.3.2 Registros con arreglos.

Los registros con arreglos tienen por lo menos un campo que es un arreglo. Por ejemplo:

```
struct  s_alumno{
  char          nombre[30];
  long int      matricula;
  float         calificaciones[6];
};
```

En la Figura III-14 se muestra el diagrama de este registro en la memoria, nótese que el arreglo `calificaciones` es uno de los campos, por lo tanto está dentro del registro `s_alumno`.

Figura III-14 *Mapa de memoria de la variable de tipo registro* `s_alumno`

Ejemplo 3.9.- Como pedirías los datos para este registro?

Solución:

```
  char cr[2]; //para limpiar el buffer
  cout << "cual es tu nombre completo?";
  cin.getline( Evaristo.nombre,30);
  cin.getline (cr,2) ;
  cout << " tu matricula?";
  cin >> Evaristo.matricula;
  for( int i= 0; i<6; i++ ){
    cout << "calificacion de la materia "
         << i+1 << " =? ";
    cin >> Evaristo.calificaciones[i];
  };
```

En la Figura III-15 se muestra la prueba de escritorio para solicitar los datos del registro `Evaristo` que es de tipo `s_alumno`. Del lado izquierdo lo que aparece en pantalla y del lado derecho lo que quedaría en memoria.

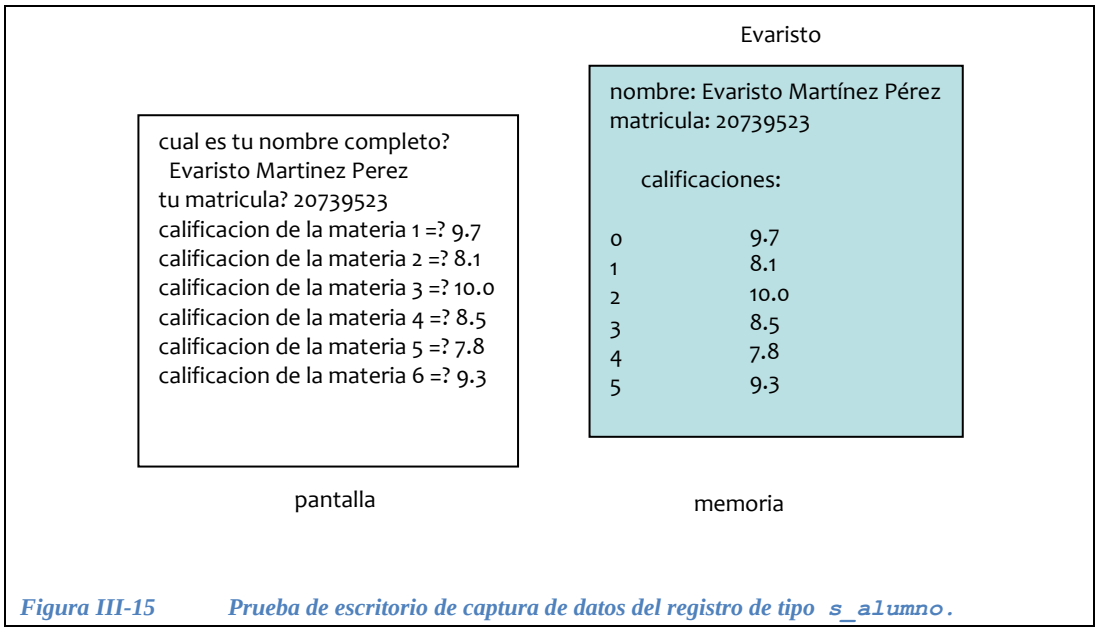


Figura III-15 *Prueba de escritorio de captura de datos del registro de tipo* `s_alumno`.

Ejercicio 3.12.- Hacer el código para obtener le promedio de las calificaciones de Evaristo.

Solución:

```
// Programa que tiene un registro con arreglo
#include<iostream.h>

struct s_alumno{
  char        nombre[30];
  long int    matricula;
  float       calificaciones[6];
};

float suma=0;
s_alumno Evaristo;
char cr[2];

main() {
  cout<< "cual es tu nombre completo?";
  cin.getline( Evaristo.nombre,30);
  cout<< " tu matricula?";
  cin>> Evaristo.matricula;

  for( int i=0; i<6; i++ ){
    cout << "calificación de la materia " << i+1 << " =? ";
    cin >> Evaristo.calificaciones[i];
    suma = suma + Evaristo.calificaciones[i];
  };
  cout<< "hola " << Evaristo.nombre << "\n tu matricula es:"
      << Evaristo.matricula << "\n y tu promedio es:" << suma/6;

  return 0;
}
```

III.3.3 Registros anidados.

En los registros anidados al menos un campo del registro es de tipo arreglo. Por ejemplo:

```
struct   s_direccion {
   char  calle[20];
   int   numero;
   char  colonia[30];
};

struct  s_alumnos {
  char         nombre[30];
  long int     matricula;
  s_direccion  domicilio;
};
```

Primero declaramos el registro `s_direccion` para poder usarlo en `s_alumnos`.

Si declaramos la variable `alumno` de tipo `s_alumnos` en la memoria tendríamos lo que se muestra en la Figura III-16:

```
s_alumnos alumno;
```

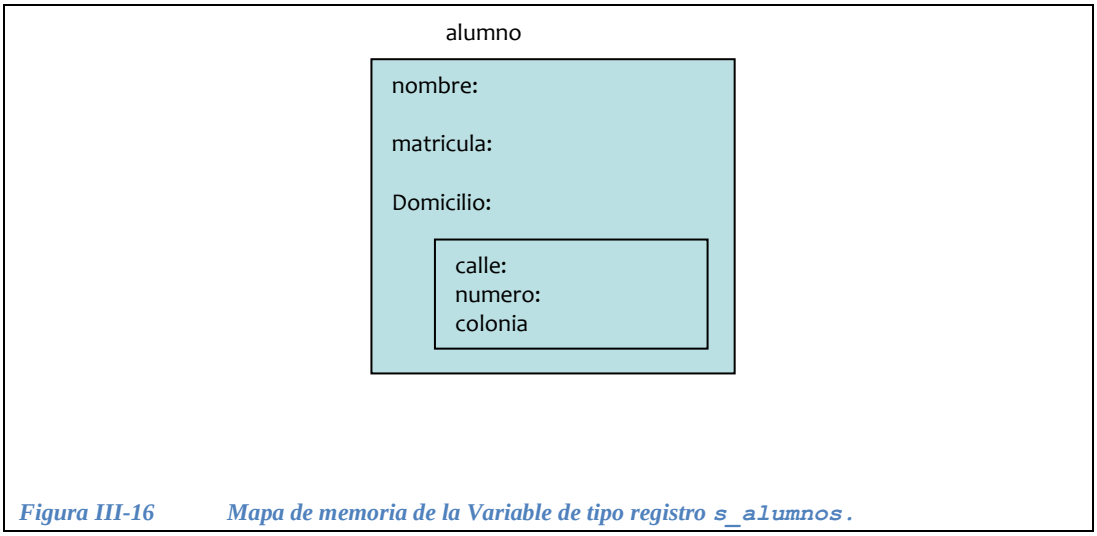


Figura III-16 *Mapa de memoria de la Variable de tipo registro `s_alumnos`.*

Ejemplo 3.10.- Como pedirías los datos para este registro?

Solución:

```
char cr[2];
cout<< "cual es tu nombre?";
cin.getline( alumno.nombre,30);
cout<< " tu matricula?";
cin>> alumno.matricula;
cin.getline(cr,2);
cout<< "tu calle?";
cin.getline(alumno.domicilio.calle,20);
cout<< "el numero de la casa?";
cin>> alumno.domicilio.numero;
cin.getline(cr,2);
cout<< "tu colonia?";
cin.getline(alumno.domicilio.colonia, 30);
```

En la Figura III-17 se muestra la prueba de escritorio para solicitar los datos del registro `alumno` que es de tipo `s_alumnos`. Del lado izquierdo lo que aparece en pantalla y del lado derecho lo que quedaría en memoria.

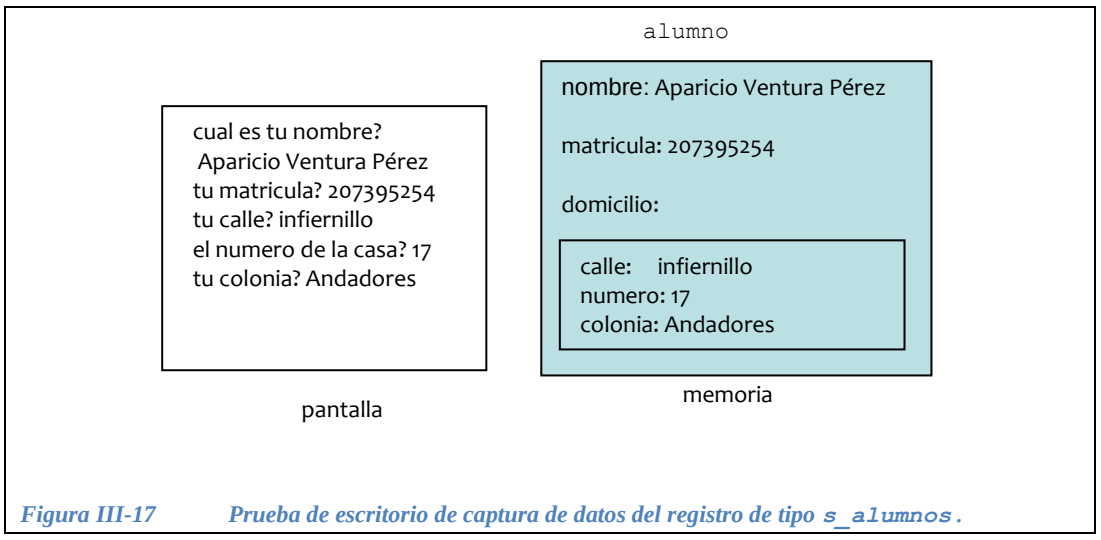


Figura III-17 *Prueba de escritorio de captura de datos del registro de tipo* `s_alumnos`.

Ejemplo 3.11.- Hacer el código para desplegar los datos de este registro.

Solución:

```
cout << alumno.nombre << " tu vives en: \n"
      << alumno.domicilio.calle << " numero: "
      << alumno.domilicio.numero<< endl
      << " colonia: " << alumno.domicilio.colonia
      << " \n y tu matricula es: "<< alumno.matricula;
```

Con el código anterior, veremos en pantalla:

Aparicio Ventura Pérez tu vives en:

infiernillo numero 17

Capítulo IV Apuntadores y acceso dinámico a memoria

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

Comprender los conceptos y elaborar programas en C++ que contengan:

- Apuntadores.
- Manejo dinámico de memoria.

IV.1 Apuntadores

IV.1.1 Concepto de apuntador

Un **apuntador** es un dato que indica la posición de otro dato. Los apuntadores se representan gráficamente con flechas. Físicamente, los apuntadores son *direcciones de memoria* en la computadora, y se guardan en variables de tipo apuntador.

En *la Figura IV-1* tenemos la variable `x` que es de tipo entero y guarda el número 1024 y la variable `y` que también es un entero y guarda el número 12. Podemos observar que la variable `p` es de tipo apuntador a entero y guarda una

dirección de memoria (representada por la flecha) que *apunta* hacia la posición del dato contenido en x.

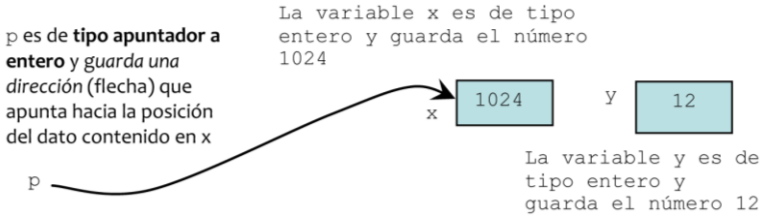


Figura IV-1 *Concepto de apuntador.*

Como a cualquier variable, a `p` se le puede cambiar el valor para que apunte hacia otro dato, esto se muestra en la Figura IV-2 en donde ahora `p` apunta al contenido de la variable `y`.

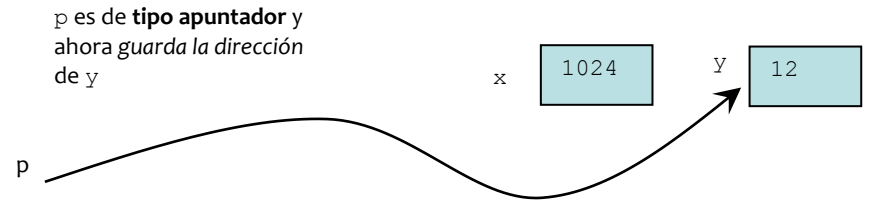


Figura IV-2 *El apuntador `p` ahora apunta a la variable `y`.*

IV.1.2 Declaración de variables de tipo apuntador

En lenguaje C y C++, Se usa el símbolo * antes del nombre de la variable para hacerla de tipo apuntador, y se debe definir a que tipo de dato apunta.

Ejemplos 4.1.-

```
// declaracion de ip como apuntador a entero:
int *ip;  // se lee como: int "pointer" ip

// declaracion de dp como apuntador a double:
double *dp; // double "pointer" dp

// declaracion de dat como apuntador a apuntador a float:
float **dat; // float "pointer" "pointer" dat
```

Ejemplos 4.2.- Declarar los siguientes apuntadores:

a) p como apuntador a long int
b) s como apuntador a char
c) nums_p como apuntador a apuntador a int
d) aux como apuntador a s_registro

Solución:

```
long int *p;
char *s;
int **nums_p;
s_registro *aux;
```

IV.1.3 Asignación de un valor a variables de tipo apuntador

A los apuntadores se les debe asignar la dirección de memoria en la que se encuentra el dato al que va a "apuntar". Para hacer lo anterior, primero se debe obtener la *dirección* del dato al que la variable de tipo apuntador va a apuntar.

El operador **&** obtiene la dirección de una variable.

Gráficamente, esta dirección es la "flecha" que apunta al contenido de la variable.

Ejemplos 4.3.-

```
p1 = &x; // a p1 se le asigna la dirección de x
p2 = p1; // a p2 se le asigna el valor de p1 (la dirección de x)
p_edad = &edad; // a p_edad se le asigna la direccion de edad
pp_x = &p1; // a pp_x se le asigna la dirección de p1
```

Estos ejemplos pueden representarse gráficamente como en la Figura IV-3.

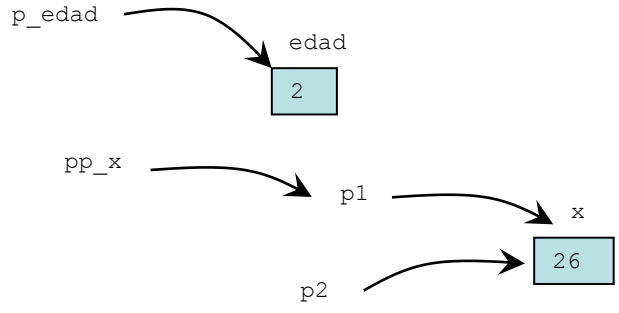


Figura IV-3 *A donde apuntan los apuntadores p1, p2, p_edad y pp_x.*

Ejemplos 4.4.-

a) Hacer que p apunte hacia m
b) Hacer que p apunte hacia aux
c) Hacer que empleado.sueldo apunte a cheque
d) Hacer que aux apunte hacia donde apunta inicio
e) Hacer que pp apunte hacia inicio.

Solución:

```
a) p =&m;
b) p =&aux;
c) empleado.sueldo = &cheque;
d) aux = inicio;
e) pp = &inicio;
```

IV.1.4 Obtención del dato al cual apunta un apuntador

Para "seguir" la flecha de una variable de tipo apuntador y obtener acceso a aquello a lo que apunta, se usa el operador ***.**

Ejemplos 4.5.-

a).- Si primero declaramos las siguientes variables y los siguientes apuntadores:

Variables:

```
int a= 700;
int c= 3, d=-1, e= 5;
int q, R1, R2, nivel;
float pago, b= 5780.50;
struct s_empleado{
   char nombre[40];
    float *sueldo;   // el apuntador es un campo del registro
};
s_empleado empleado;
```

Apuntadores:

```
int *x, *y, *z, *p; // x, y, z, p: son apuntadores a entero

int **s; // s es un apuntador que apunta a un apuntador que apunta
         // a un  entero
```

b).- Luego hacemos que los apuntadores apunten a las variables que se indican a continuación:

```
P = &a;
empleado.sueldo = &b;
x = &c;
y = &d;
z = &e;
```

Entonces las siguientes asignaciones pueden representarse gráficamente como en la Figura IV-4.

```
q = *p;    // a q se le asigna "lo que apunta" p
pago = *empleado.sueldo; //el apuntador puede estar dentro de
                         // un registro
R1 = *x + *y + *z;
nivel = **s; // s es un apuntador a apuntador
             // nivel = lo que apunta lo que apunta s
R2 = *x * *y * *z; //  El primer * es multiplicación y el
                   //  segundo lo que apunta y
```

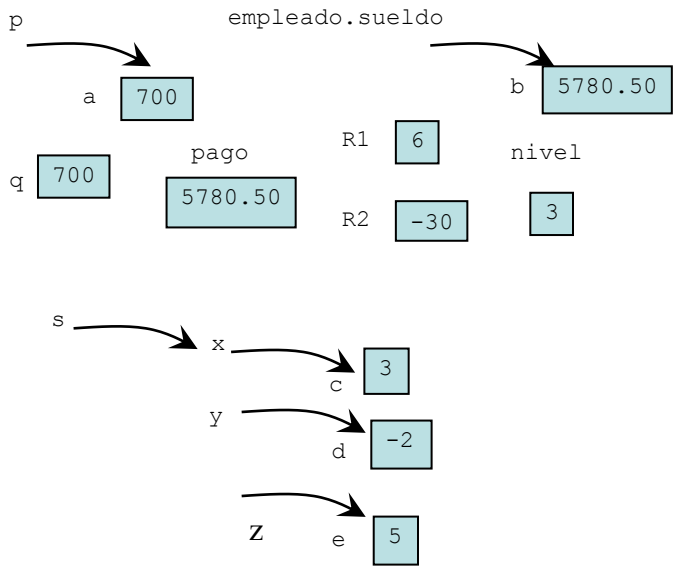


Figura IV-4 *Representación gráfica de variables y apuntadores del ejemplo 4.3 .*

Si hacemos la siguiente modificación, entonces `y` apunta a `c`, como se muestra en la Figura IV-5.

```
y = *s; // "y" apuntará a donde apunta lo que apunta "s",
        // es decir,
        // "y" apuntará a donde apunta "x", en conclusión:
        // "y" apuntará a "c"
```

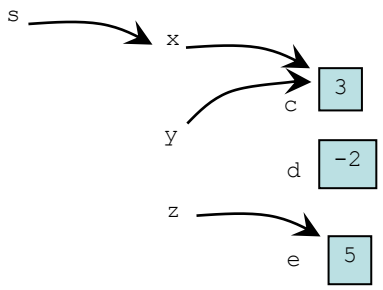


Figura IV-5 *El apuntador y apunta a la variable c.*

Ejemplo 4.6.-

1.- Si tenemos:

```
float saldo, *salida;
float cantidad = 2403.25;
salida = &cantidad; /salida apunta a la variable "cantidad"
```

Asignar a `saldo` el dato apuntado por `salida`. ¿Qué cantidad habrá en `salida`?

2.- Si tenemos:

```
int a = 17, b = 25;
int *x, *y;
x = &a; // "x" apunta a la variable "a"
y = &b; // "y" apunta a la variable "b"
```

Hacer que la suma de lo que apunta `x` más lo que apunta `y` se escriba en pantalla.

3.- Si tenemos:

```
double *salida, saldo = 34.72, num;
double **p; // "p" es una apuntador que apunta a un
            // apuntador a double
```

Asignar a `num` el dato apuntado por el apuntador apuntado por `pp`

```
saldo = *salida;  // a saldo se le asigna lo que apunta salida
cout << *x + *y;  // escribir lo que apunta x más lo que apunta y
num = **pp;  // a num se le asigna lo que apunta lo que apunta pp
```

IV.1.5 Asignación de un valor a lo que apunta un apuntador

Los apuntadores pueden ser utilizados también para modificar el contenido del dato al que apuntan.

Ejemplo 4.7.-

```
// modificación del contenido de los datos por un apuntador
#include <iostream.h>

main() {
  int i,j,k, *p;

  p = &i;     // p apunta a i
  *p = 3;     // se le asigna 3 a lo que apunta p
  p = &j;     // p apunta a j
  *p = 30;    // se le asigna 30 a lo que apunta p
  p = &k;     // p apunta a k
  *p = 300;   // se le asigna 300 a lo que apunta p

  cout << "i = "<< i << " j= " << j " k = " << k;

  return 0;
}
```

¿Cual es el resultado en pantalla de este programa?

Respuesta

```
i = 3   j = 30   k = 300
```

IV.1.6 Algunos puntos adicionales sobre apuntadores

IV.1.6.1 Comparando "dirección de", "lo que apunta" y "contenido de"

Declaremos la variable `x` como una variable de tipo entero que contiene el valor 9:

```
int x=9;
```

y el apuntador a entero `ptr_int`:

```
int *ptr_int;
```

Ahora hagamos que `ptr_int` apunte a `x`:

```
ptr_int = &x;  // ptr_int contiene la dirección de x
```

Lo anterior se puede apreciar gráficamente en la Figura IV-6:

Figura IV-6 *El apuntador `ptr_int` apunta a la variable `x` que contiene el valor 9 .*

En este caso, si damos la instrucción:

```
cout << ptr_int;
```

obtendremos en la pantalla ***el contenido de*** `ptr_int` que es ***la dirección de*** `x`. `0x0012F320` (es una dirección particular que depende de cada computadora)

Si damos la instrucción:

```
cout << *ptr_int;
```

obtendremos en la pantalla ***lo que apunta*** `ptr_int` que es ***el contenido de*** `x:` 9

Si damos la instrucción:

```
cout << &ptr_int;
```

obtendremos en la pantalla ***la dirección de*** `ptr_int` que es diferente de ***la dirección*** de `x`.

```
0x0012F2AA
```

IV.1.6.2 Inicialización de variables de tipo apuntador.

Las variables no inicializadas contienen "basura", por ejemplo si declaramos:

```
int x, * ptr_int;
```

tanto `x` como `ptr_int` contienen "basura" y no se sabe cual es la dirección que guarda `ptr_int`, sin embargo, si inicializamos `x` con 9 y a `ptr_int` con la dirección de `x:`

```
int x=9, * ptr_int= &x;
```

Ahora `ptr_int` contiene la dirección de `x` como en la Figura IV-6.

Cuando no se inicializa un apuntador, puede ser que la "basura" que contiene apunte a una dirección utilizada por otro programa (que puede ser incluso el sistema operativo).

Si en el programa se cambia *lo que apunta* este apuntador, puede ocasionarse una falla grave en el sistema.

El compilador no puede detectar este tipo de defectos en el programa así que es responsabilidad del programador el inicializar siempre todos los apuntadores de manera que apunten hacia variables conocidas antes de usarlos.

En el siguiente programa se comete un error al no inicializar el apuntador "p" y modificar el dato al que apunta:

```
int   a=10, b, *p, *q;
q = &a;    // "q" apunta a la variable "a".
b = *q;    // se le asigna a "b" lo que apunta "q" (que es "a")
*p = 20;   // error: asignación no valida, ¿a donde apunta "p"?
```

IV.1.6.3 El apuntador nulo NULL

NULL es un valor constante que se define en varios de los archivos de encabezado que se utilizan normalmente (stdio.h, stdlib.h, string.h) de la siguiente manera:

```
#define NULL 0
```

Cuando una variable de tipo apuntador tiene un valor igual a NULL ésta apunta a "ninguna parte", también se dice que "apunta a tierra".

Por disciplina de programación, a las variables de tipo apuntador que no están siendo usadas se les debe asignar el valor NULL para indicar así que no contienen una dirección válida.

Antes de usar la dirección contenida en una variable de tipo apuntador es importante primero compararla con NULL y evitar su uso es caso de que sea igual a este valor.

Cuando se inicializa un apuntador con NULL evitamos, que se hagan accesos a áreas desconocidas de memoria, las cuales podrían estar siendo usadas por otros programás, este caso podría darse cuando se usa un apuntador sin haberle asignado antes una dirección válida. Prevenir estos casos es importante, ya que estas fallas pueden ser muy graves.

IV.1.6.4 Compatibilidad de apuntadores

No todos los apuntadores son compatibles, es decir, se debe tener cuidado al trabajar con apuntadores que *apuntan a* tipos diferentes de datos. No es lo mismo un apuntador a `int` que un apuntador a `float`. No es posible asignar tipos distintos de apuntadores.

Ejemplo 4.8.- Si tenemos los siguientes apuntadores:

```
int *x, *y, **z;
float *u, *v, **w;
```

los cuales podemos apreciar gráficamente en la Figura IV-7:

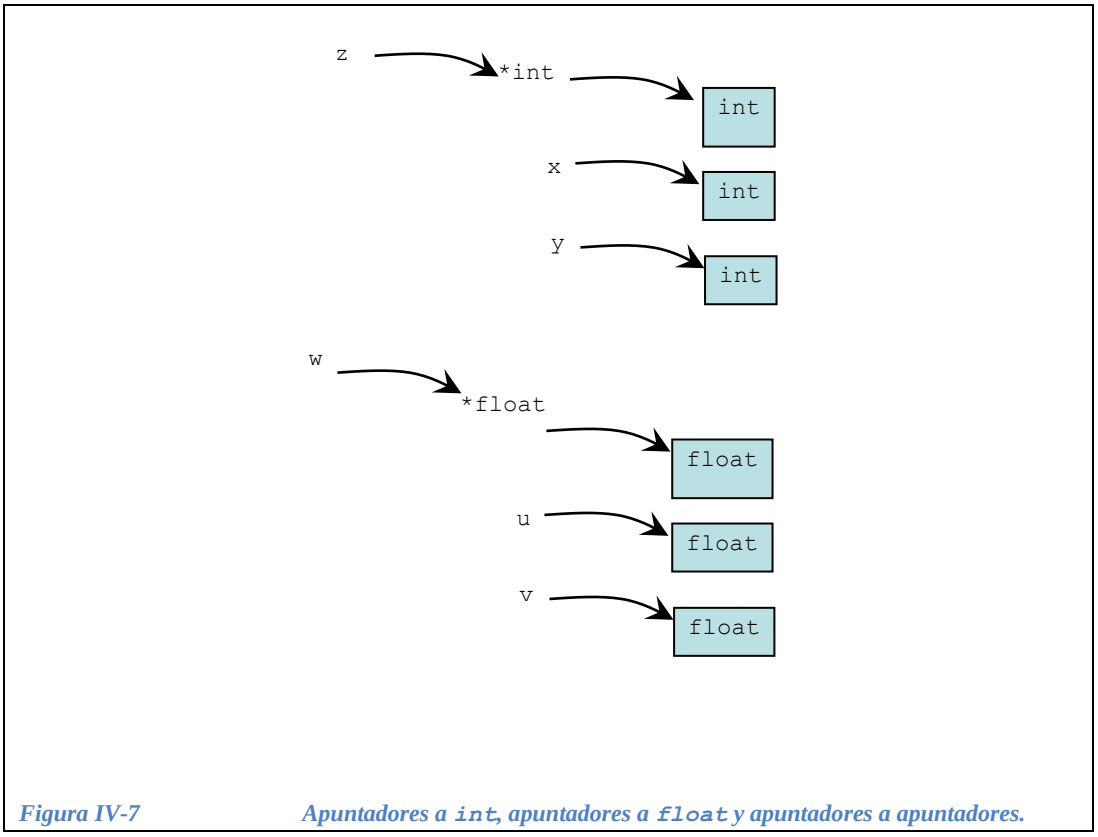


Figura IV-7 *Apuntadores a `int`, apuntadores a `float` y apuntadores a apuntadores.*

A continuación se muestran cuales son las operaciones válidas para los apuntadores de la Figura IV-7 considerando que previamente cada uno de ellos apunta a una dirección en particular.

```
x = y;   // es válido, la dirección que contiene "y"
         // ahora la contiene "x"

*z = x;  // válido, lo que apunta *z(a un int) se carga con
         // el entero al que apunta "x"

**z = *y // es válido, el entero al que apunta **z se carga con
         // el entero al que apunta "y"

y = *z; // es válido, "y" se carga con la dirección que contiene
        // el apuntador a int

*w = v; // es válido, el apuntador a float apuntará al mismo lugar
        // que "v" que también es un apuntador a float

x = z; // no es válido,"x" apunta a un int mientras que "z" apunta
       // a un apuntador a int

u = x; // no es válido, "u" contiene la dirección de un float,
       // mientras que "x" contiene la dirección de un int

z = w; // no es válido "z" contiene la dirección de un apuntador a
       // int, mientras que "w" contiene la dirección un apuntador
       // a float

*x = *z; // no es válido un entero no se puede cargar con un
         // apuntador

**w = **z; // es válido a un float si se le puede asignar un
           // entero
```

IV.1.7 Ejercicios de apuntadores

Ejercicio 4.1.- Codificar las siguientes instrucciones:

- Declarar una variable de tipo entero llamada `a`.
- Declarar un apuntador a entero llamado `ptr`.
- Asignar a `ptr` la dirección de `a`.
- Asignar a lo que apunta `ptr` el número 24.
- Escribir en pantalla el valor de `a`.
- Escribir en pantalla el contenido del apuntador `ptr`.
- Escribir en pantalla lo apuntado por `ptr`.
- ¿Qué valor es el que sale en la pantalla?

Primera *Solución parcial*:

El siguiente programa contiene la solución de los primeros 4 puntos anteriores. ¿Qué valor resulta para la variable `a`?

```
// ejercicio 1 de apuntadores
#include <iostream.h>

main() {
  int a;
  int *ptr;

  ptr = &a;
  *ptr = 24;

  // Aqui imprimir lo que contiene a

  // Aquí imprimir el contenido del apuntador ptr

  // Aquí imprimir lo que apunta ptr

  return 0;
}
```

Solución completa:

```
// ejercicio 1 de apuntadores
#include <iostream.h>

main() {
  int a;
  int *ptr;

  ptr = &a;
  *ptr = 24;

  // Aqui imprimir lo que contiene a
  cout << "a contiene: " << a;

  // Aquí imprimir el contenido del apuntador ptr
  cout << "\n el apuntador contiene: " << ptr;

  // Aquí imprimir lo que apunta ptr
  cout << "\n el apuntador apunta a: " << *ptr;

  return 0;
}
```

Lo que debe salir en la pantalla es lo siguiente:

```
a contiene:  24
el apuntador contiene: dirección de memoria (asignada por el SO)
el apuntador apunta a: 24
```

Ejercicio 4.2.- Codificar las siguientes instrucciones:

- Declarar 3 variables de tipo apuntador a entero llamadas `a, b` y `c` y dos enteros llamados `i` y `j`.
- Hacer que `a` y `b` apunten a `i` y que `c` apunte a `j`.
- Asignar a lo que apunta `b` un 4 y a lo que apunta `c` un 3.
- Asignar a lo que apunta `a` la suma del doble de lo que apunta `b` más lo que apunta `c`.

- Escribir en pantalla `i` y `j`.
- ¿Qué valores salen en la pantalla?

Primera Solución parcial. Completar el siguiente programa ¿Qué valor resulta para las variables *i, j* ?

```
// ejercicio 2 de apuntadores
#include <iostream.h>

main() {
  int *a, *b, *c;
  int i, j;

  // Hacer que a y b se carguen con la dirección de i

  // Hacer que c apunte a j

  // poner un 4 en donde apunta b

  // poner un 3 en donde apunta c

  *a = 2**b+*c;

  // desplegar los valores quedaron en las variables i y j


  return 0;
}
```

Solución completa:

```
// ejercicio 2 de apuntadores
#include <iostream.h>

main() {
  int *a, *b, *c;
  int i, j;

  // Hacer que a y b se carguen con la dirección de i
  a = b = &i;

  //  Hacer que c apunte a j
  c = &j;

  // poner un 4 en donde apunta b
  *b = 4;

  // poner un 3 en donde apunta c
  *c = 3;

  *a = 2**b+*c;

  // que valores quedaron en las variables i y j?
  cout << "\n i= " << i << " j= " << j;

  return 0;
}
```

El resultado en la pantalla es el siguiente:

i=11 j=3

Ejercicio 4.3.- Codificar las siguientes instrucciones:

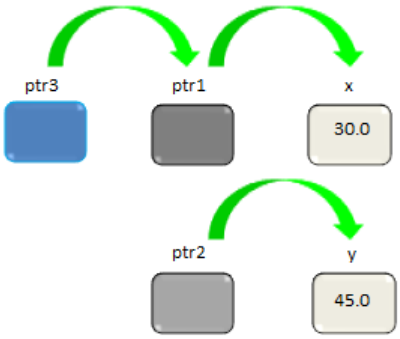
- Declarar 2 variables de tipo `double` llamadas `x` y `y` e inicializarlas con los valores 30 y 45 respectivamente.
- Declarar `ptr1` y `ptr2` como apuntadores a `double` e inicializarlos con las direcciones de `x` y `y` respectivamente.
- Declarar `ptr3` como apuntador a apuntador a `double` e inicializarlo con la dirección de `ptr1`
- Asignar a `x` la suma de lo que apunta `ptr1` más `x`
- Asignar a lo que apunta `ptr2` la suma del triple de `y` más el doble de lo que apunta lo que apunta `ptr3`
- Asignar a lo que apunta `ptr3` la dirección de `y`
- Asignar a `ptr3` la dirección de `ptr2`
- Asignar a lo que apunta `ptr3` la dirección de `x`
- Asignar a lo que apunta lo que apunta `ptr3` la diferencia de lo que apunta `ptr2` menos el producto de lo que apunta `ptr1` por `x`
- Escribir `x` y `y`.
- ¿Qué valores salen en la pantalla?

Solución:

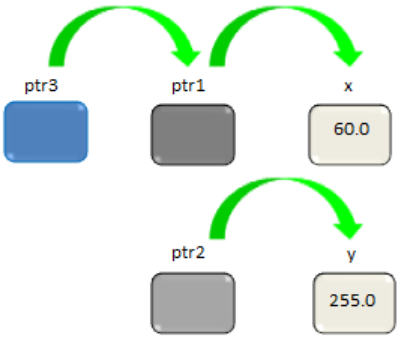
```
#include <iostream.h>

main() {
  double x=30, y=45;
  double * ptr1, *ptr2;
  double ** ptr3;

  ptr1=&x; ptr2=&y;  // ptr1 apunta a "x"  y ptr2 apunta a "y"
  ptr3=&ptr1;        // ptr3 apunta a "ptr1"

  x = *ptr1 + x;            // x= 30+30 = 60
  *ptr2 = 3*y + 2***ptr3;   // y= 3*45 + 2*60 = 255
  *ptr3 = &y;               // ahora ptr1 apunta a y
  ptr3 = &ptr2;             // ptr3 apunta a ptr2
  *ptr3 = &x;               // ptr2 apunta a "x"
  **ptr3 = *ptr2 - *ptr1*x; // x= 60 - 255*60 = -15240

  cout << "x= " << x << "   y=  " << y;

  return 0;
}
```

A continuación presentamos una representación gráfica de lo que sucede en el programa anterior:

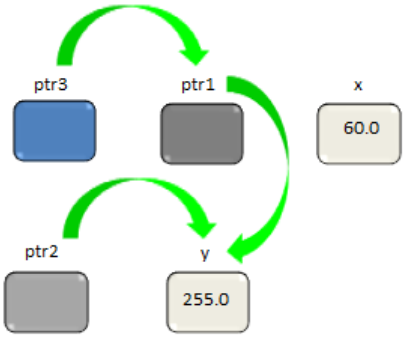
```
ptr1=&x; ptr2=&y;
// ptr1 apunta a "x"  y ptr2 apunta
a "y"

ptr3=&ptr1;
// ptr3 apunta a "ptr1"
```



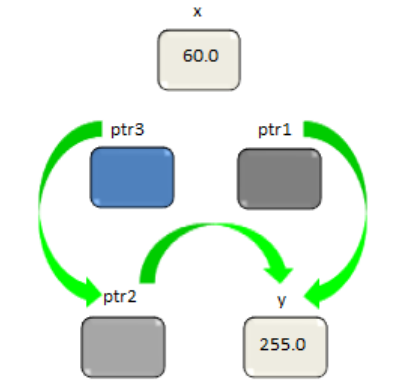

```
x = *ptr1 + x;
// x= 30+30 = 60.0

*ptr2 = 3*y + 2***ptr3;
// y= 3*45 + 2*60 = 255.0
```

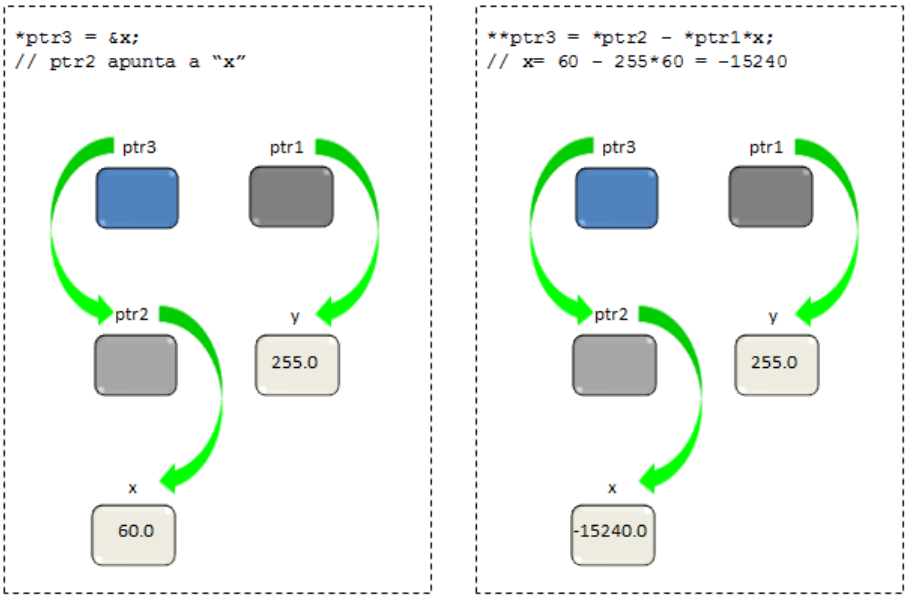



```
*ptr3 = &y;
// ahora ptr1 apunta a y
```




```
ptr3 = &ptr2;
// ptr3 apunta a ptr2
```





Por lo tanto, el resultado final es:

```
x=-15240     y=255
```

Ejercicio 4.4.- Completar el siguiente programa.

```
#include <iostream.h>

main() {
  int *pointer_int, a, x=7;
  float *pointer_float, b, y= 4.0;

  // el apuntador a entero se carga con la dirección de x
  pointer_int = . . . ;
  cout << "el contenido de x es " << . . . << endl;
  cout << "la direccion de x es: " << . . . << endl;
  cout << "pointer_int contiene: " << . . . << endl;
  cout << "pointer_int apunta a: " << . . . << endl;

  // el apuntador a float se carga con la direccion de y
  pointer_float = . . .;
  cout << "el contenido de y es " <<  . . . << endl;
  cout << "la direccion de y es: " << . . . << endl;
  cout << "pointer_float contiene: " << . . . << endl;
  cout << "pointer_float apunta a: " << . . . << endl;

  // a se carga con lo que apunta pointer_int
  a = . . .;

  // b se carga con lo que apunta pointer_float
  b= . . .;

  // desplegar a y b
  cout<< "a= " << . . .<< endl << "b= " <<. . . ;
  return 0;
}
```

Solución:

```
#include <iostream.h>

main() {
  int *pointer_int, a, x=7;
  float *pointer_float, b, y= 4.0;

  // el apuntador a entero se carga con la dirección de x
  pointer_int = &x;
  cout << "el contenido de x es " << x << endl;
  cout << "la direccion de x es: " << &x << endl;
  cout << "pointer_int contiene: " << pointer_int << endl;
  cout << "pointer_int apunta a: " << *pointer_int << endl;

  // el apuntador a float se carga con la direccion de y
  pointer_float = &y;
  cout << "el contenido de y es " << y << endl;
  cout << "la direccion de y es: " << &y << endl;
  cout << "pointer_float contiene: " << pointer_float << endl;
  cout << "pointer_float apunta a: " << *pointer_float << endl;

  // a se carga con lo que apunta pointer_int
  a = *pointer_int;

  // b se carga con lo que apunta pointer_float
  b= *pointer_float;

   // desplegar a y b
  cout<< "a= " << a << endl << "b= " << b;

  return 0;
}
```

IV.2 Alojamiento dinámico de memoria

Muchos programas requieren del uso de la memoria solo temporalmente y resulta un desperdicio de recursos el que la tengan reservada todo el tiempo. Si los programas reservan memoria solamente cuando la necesitan, permitirán que otros programas la usen sin que tengan que esperar a que otros programas terminen su ejecución.

Cuando se reserva y libera memoria durente la ejecución de los programas se utiliza por lo general el término inglés *heap memory*. Este término puede traducirse al español como "pedacería de memoria" y se puede visualizar como un montón de pedazos de memoria que han sido "liberados" por los programas que ya no la necesitan.

El sistema operativo tiene un *administrador de memoria* que consiste en un conjunto de programas que administran esta "pedacería" y asignan pedazos a los programas que soliciten memoria. El administrador de memoria puede juntar varios pedazos (continuos) para entregar a los programas pedazos grandes.

Cuando se solicita la ejecución de un programa, el sistema operativo calcula cuánta memoria necesita éste y solicita al *administrador de memoria* un pedazo suficientemente grande para asignarlo al programa. A esto se le llama *alojamiento estático* ya que esta memoria no es liberada sino hasta que el programa termina su ejecución.

Si durante la ejecución del programa éste solicita más memoria, lo debe hacer mediante solicitudes al *administrador de memoria*. A esto se le llama *alojamiento dinámico* ya que esta memoria puede ser solicitada y liberada por el mismo programa cuando quiera durante su ejecución. Ver Figura IV-8.

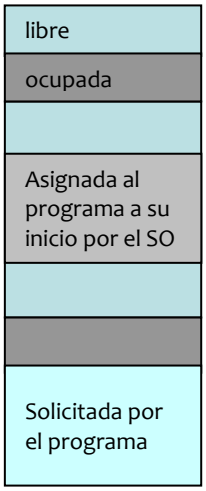


Figura IV-8 ***El administrador de memoria otorga memoria estáticamente a un programa para que éste se ejecute y, dinámicamente, una vez que éste se está ejecutando y requiere de más memoria.***

Una vez que el programa ya no necesita la memoria debe liberarla al montón de pedacería (*heap memory*) a través también del *administrador de memoria*. Si no se libera, podría quedar bloqueada indefinidamente. Finalmente, cuando el programa termina su ejecución, el SO libera la memoria que fue le fue asignada al inicio.

IV.2.1 Alojamiento dinámico de memoria en C++

Para solicitar memoria dinámicamente, el programa llama al administrador de memoria solicitándole un bloque de memoria continuo de un cierto tamaño. En C++ se hace con el operador **`new`** agregando el *tipo* de variable que se desea alojar:

```
new <tipo>;
```

El administrador de memoria debe saber el tamaño de la memoria que se requiere, así como el nombre del apuntador en donde va a guardar la *dirección de inicio* de la memoria que otorgó. Así por ejemplo, la instrucción:

```
ptr_int = new int;
```

Guarda en el apuntador `ptr_int` la dirección del entero que alojó en la memoria dinámica.

El resultado de un alojamiento en memoria dinámica es un apuntador hacia el inicio del bloque de memoria asignado.

Si el resultado es NULL significa que no se alojó la memoria solicitada por alguna causa.

Es posible dar un valor inicial a la variable que se aloja en la memoria dinámica, por ejemplo:

```
ptr_double = new double(0.0); // declaración e inicializacion
                              // del double
```

En realidad, alojar un número entero o un número real en la memoria dinámica no tiene mucho sentido, uno de los casos en los que el alojamiento de memoria dinámica cobra sentido, es cuando se trabaja con arreglos de tamaño variable, esto se hace de la siguiente manera:

```
new <tipo>[ <número> ];
```

Desde luego también es necesario guardar la dirección de inicio en un apuntador, a continuación se dan algunos ejemplos:

```
int * arr_int;  // se declara el apuntador a entero en donde se
                // guardará la dirección de inicio del arreglo
arr_int = new int[20]; // arreglo de 20 enteros
```

También es posible hacer un arreglo de registros de tamaño variable. Como ejemplo tenemos:

```
struct empleado {
   int    num_empleado;
   char   nombre[40];
   float  sueldo;
};

s_empleado *empleados; // se declara un apuntador al registro
empleados = new s_empleado[n]; // arreglo de n registros
```

IV.2.2 Liberación dinámica de memoria en C++

Para liberar la memoria que se alojó previamente mediante el operador **new**, se hace un llamado al administrador de memoria solicitándole la liberación del bloque. En C++ se hace con el operador:

```
delete <apuntador>;
```

por ejemplo:

```
delete ptr_int;
delete ptr_double;
```

o bien para liberar arreglos, con:

```
delete [] apuntador;
```

por ejemplo:

```
delete [] arr_int; // se liberan 20 enteros
delete [] empleados; // se liberan los n elementos
```

Si el apuntador usado es igual a NULL o a otra dirección que no apunta al inicio de un bloque previamente alojado habrá un error y la memoria no se libera.

Si el programa termina antes de que se desaloje la memoria, ésta no podrá ser usada por otros programás, la memoria disponible puede llegar a agotarse y entonces no se podrán ejecutar otros programa, la única manera de liberar la memoria retenida será reiniciando el sistema.

Es responsabilidad del programador asegurarse de que su programa libere toda la memoria que ocupó antes de terminar. Incluso en caso de que su ejecución termine por alguna falla.

Ejemplo 4.9.- Hacer uso de la memoria dinámica para declarar un arreglo de tamaño `n`, de `float` llamado `reales`, `n` es un número entero que se le pide al usuario. Desalojar la memoria al final.

```
// programa para alojar un arreglo de reales de tamaño variable
#include <iostream.h>

main() {
  int n;
  float *reales; // el nombre del arreglo es un apuntador

  // preguntar al usuario por el numero n
  cout << "de que tamaño es el arreglo?;
  cin >> n;

  // alojar el arreglo de tamaño n
  reales = new float[ n ];

  // liberar la memoria
  delete [] reales;
  return 0;
}
```

Ejemplo 4.10.- Hacer lo mismo que el ejemplo anterior pero ahora alojando y desalojando los siguientes arreglos:

- Un arreglo de enteros llamado `a` de tamaño `n`.
- Un arreglo de reales de tipo `double` llamado `b` del doble del tamaño de `a`.
- Un arreglo de reales de tipo `float` llamado `c` de tamaño `3n - 1`

Solución:

```
// programa para alojar varios arreglos de tamaño variable
#include <iostream.h>

main() {
  // hacer las declaraciones necesarias
  int n, *a;
  double *b;
  float *c;
  // preguntar al usuario por el numero n
  cout << "de que tamaño es el arreglo?; cin >> n;

  // alojar los tres arreglos
  a = new int[ n ];
  b = new double[ 2*n ];
  c = new float[ 3*n - 1 ];

  // liberar la memoria
  delete [] c;
  delete [] b;
  delete [] a;

  return 0;
}
```

Ejemplo 4.11: A continuación se presenta un ejemplo ilustrativo de alojamiento dinámico de memoria.

```
// ejemplo ilustrativo de alojamiento dinámico de memoria
#include <iostream.h>

main() {
  int *n = new int(100);
  if( n != NULL ) {
    double *x = new double;
    if( x != NULL ) {
      double *suma = new double(0.0);
      if( suma != NULL ) {
        for( int i=0; i<*n; i++ ) {
          cin >> *x;
          *suma = *suma + *x;
        };
        cout << *suma;
        delete suma;
      }
      else
        cout << "no se pudo alojar suma";}

      delete x;
    }
    else
      cout << "no se pudo alojar x";

    delete n;
  }
  else
    cout << "no se pudo alojar n";
  }
  return 0;
}
```

n apunta a un entero cuyo valor inicial es 100

Cada alojamiento debe checarse!

Los alojamientos que sí se lograron deben liberarse.

IV.2.3 Ejercicios de alojamiento dinámico de memoria

Ejercicio 4.5.- Completar el siguiente programa que aloja un arreglo de `n` estructuras de la siguiente forma:

```
struct s_materia{
  char  nombre[30];
  float calific;
};
```

El programa pregunta al usuario por el tamaño del arreglo `n`. Captura los datos y los despliega en pantalla. Después desaloja la memoria correspondiente.

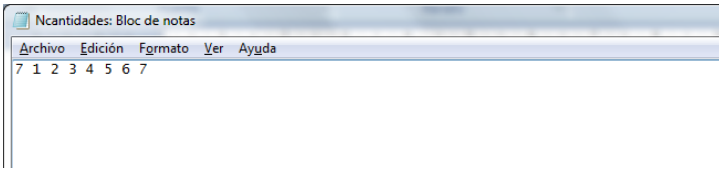
```
// arreglo variable de estructuras
#include<iostream.h>

main() {
  int  n,i;
  char cr[2];
  struct s_materia{
    char  nombre[30];
    float calific;
  };
  // declarar un apuntador a la estructura s_materia

  // preguntar por el tamaño del arreglo

  // intento de alojamiento de memoria

  if( materias == NULL )
           // alojamiento fallido

 // pedir datos
  for(    . . .      ) {
     cin.getline(cr,2); // elimina basura por el último cin
     cout << "Cuál es el nombre de la materia " << i+1 << "?";
     cin.getline(materias[i].nombre,30);
     cout << " la calificación en esa materia?";
     cin >> . . . . . ;
  };
  // desplegar los datos
  for( i=0; i<n; i++ ) {
    cout << "En " << . . . << " tienes: "
         << . . . << endl;
  };
  // desalojar memoria

  return 0;
}
```

Solución:

```
// arreglo variable de estructuras
#include<iostream.h>

main() {
  int  n,i;
  char cr[2];
  struct s_materia{
    char  nombre[30];
    float calific;
  };

  // declarar un apuntador a la estructura s_materia
  s_materia *materias;

  // preguntar por el tamaño del arreglo
  cout << "tamaño del arreglo?";
  cin >> n;

  // intento de alojamiento de memoria
  materias = new s_materia[n];

  if( materias == NULL )
    return -1;  // alojamiento fallido

  // pedir datos
  for( i=0; i<n; i++ ) {
     cin.getline(cr,2); // elimina basura por el último cin
     cout << "Cuál es el nombre de la materia " << i+1 << "?";
     cin.getline(materias[i].nombre,30);
     cout << " la calificación en esa materia?";
     cin >> materias[i].calific;
  };

  // desplegar los datos
  for( i=0; i<n; i++ ) {
    cout << "En " << materias[i].nombre << " tienes: "
         << materias[i].calific << endl;
  };

  // desalojar memoria
  delete [] materias;

  return 0;
}
```

Ejercicio 4.6.- Con "Block de Notas" preparar un archivo de texto, el primer dato debe contener el número de datos del arreglo, y a continuación, separados por espacios, los datos del arreglo. El archivo debe estar en el mismo directiorio donde se guarde el proyecto, llamaremos al archivo de texto "`Ncantidades.txt`".

Por ejemplo:



Completar el programa para que lea todas las cantidades del archivo y las almacene en un arreglo de reales.

Hacer que el programa escriba en pantalla todas las cantidades multiplicadas por dos, y en orden inverso al que fueron leídas.

Para el ejemplo del archivo anterior, en DevCpp, la salida sería:

Solución parcial (en DevCpp):

```
#include <cstdlib>
#include <iostream>
// leer n cantidades y escribirlas
// en orden inverso
#include <fstream>

using namespace std;

int main( int argc, char *argv[] ) {
  ifstream entrada;
  entrada.open(". . . ");
  if( !entrada.is_open() ) {
    cout << "no se pudo abrir el archivo" << endl;
    system("PAUSE");
    return -2;

  };

  int n;
  entrada >> . . . ;
  double *cantidades;
  cantidades =. . . . ;

  if( cantidades == NULL ) {
    cout << "no se pudo alojar memoria suficiente"
         << endl;
    entrada.close();
    system("PAUSE");
    return -1;
  };

  for( int i=0; i<n; i++ )
    entrada >> . . . . ;

  entrada.close();

  for( int i=… ; i>=0; … )
    cout << "cantidades ["<< i << "]=  "
         << . . .
         << endl;

  delete [] cantidades;

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Solución completa (en DevCpp):

```
#include <cstdlib>
#include <iostream>
// leer n cantidades y escribirlas
// en orden inverso
#include <fstream>

using namespace std;

int main( int argc, char *argv[] ) {
  ifstream entrada;
  entrada.open("Ncantidades.txt");
  if( !entrada.is_open() ) {
    cout << "no se pudo abrir el archivo" << endl;
    system("PAUSE");
    return -2;
  };

  int n;
  entrada >> n;
  double *cantidades;
  cantidades = new double[n];

  if( cantidades == NULL ) {
    cout << "no se pudo alojar memoria suficiente"
         << endl;
    entrada.close();
    system("PAUSE");
    return -1;
  };

  for( int i=0; i<n; i++ )
    entrada >> cantidades[i];
  entrada.close();

  for( int i=n-1; i>=0; i-- )
    cout << "cantidades ["<< i << "]=  "
         << 2* cantidades[i]
         << endl;

  delete [] cantidades;

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

IV.3 Arreglos multidimensionales dinámicos

IV.3.1 Diferencia entre arreglos estáticos y dinámicos

.Los arreglos pueden ser de dos o más dimensiones, por ejemplo, una matriz tiene dos dimensiones: los renglones y las columnas, de tal forma que una matriz `A` de enteros de tamaño `n x m` (donde n y m son constantes) se define así:

```
int   A[n][m];
```

Por ejemplo, en la Figura IV-9 se ilustra una matriz `B` con cinco filas y cuatro columnas.

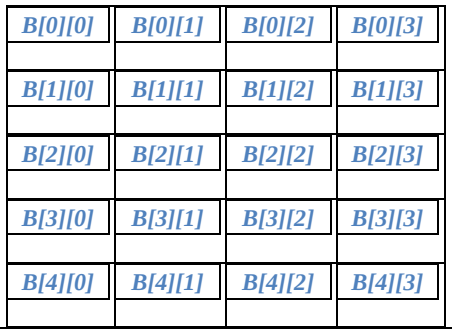


Figura IV-9 *Ejemplo de una matriz (arreglo bidimensional).*

Cuando se trabaja con memoria estática, se debe determinar un tamaño constante del arreglo, como en el siguiente ejemplo que captura los datos de una matriz de enteros llamada `matriz` la cual tiene seis filas y cinco columnas.

```
int matriz[6][5];
for( int i = 0; i < 6; i++ ){ // para cada fila
  for( int j = 0; j < 5; j++ ){ // para cada columna
    cout << "matriz[" << i << ", " << j << "] = ?";
    cin >> matriz[i][j];
  };
  // En este ciclo se capturaron las columnas de la fila i
};
```

El alojamiento dinámico de memoria brinda la posibilidad de trabajar con arreglos de tamaño variable. Si tenemos una matriz de enteros llamada A y queremos manejarla con memoria dinámica, es necesario declarar su nombre como un doble apuntador a entero, solicitar al usuario el tamaño de la matriz y después alojar la memoria.

```
int **A;
cout << "numero de renglones de A?"; cin >> n;
Cout << "numero de columnas de A? "; cin >> m;

A = new (int *)[n]; // arreglo de apuntadores a int
for( int i=0; i<n; i++ )
  A[i]= new int[m];
```

En la Figura IV-10 se ilustra la forma en la que se aloja la memoria para la matriz `A`.

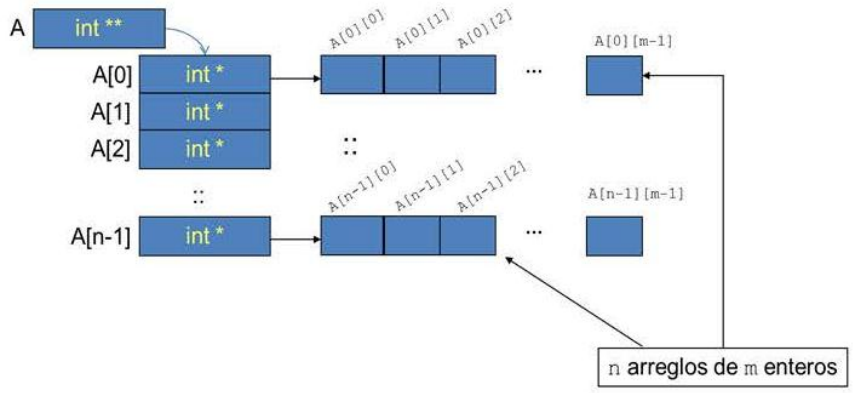


Figura IV-10 Alojamiento en memoria de una matriz.

Para capturar los datos de una matriz de enteros llamada `matriz` la cual tiene seis filas y cinco columnas, pero trabajando ahora con memoria dinámica, tenemos:

```
int **matrizA;
matrizA = new (int *)[6]; // alojar arreglo de apuntadores
                          //  a int
for( int i=0; i<6; i++ )
  matrizA[i] = new int[5]; // alojar 6 arreglos de 5 int c/u

for( int i=0; i<6; i++ ) { // para cada fila
  for( int j=0; j<5; j++ ) { // para cada columna
    cout << "matrizA[" << i << ", " << j << "] = ?";
    cin >> matrizA[i][j]; // igual que con memoria estática
  };
};
```

Y, para liberar la memoria, primero se desalojan los `n` arreglos de los renglones y luego el arreglo de arreglos, de la siguiente forma:

```
if( A != NULL ) {
  for( i=0; i<n; i++ )
    if( A[i] != NULL )
      delete [] A[i];
  delete [] A;
};
```

Ejemplo 4.12.- Hacer un programa que capture los datos de una matriz de 30 renglones (filas) y 40 columnas y que los despliegue en pantalla utilizando memoria estática.

En seguida se ilustra el programa que realiza lo anterior, puede apreciarse que las matrices se recorres utilizando dos ciclos `for` anidados, con el primer `for` se recorren los renglones, y para cada renglón se recorren las columnas con el segundo `for` anidado. Nótese que mientras que en matemáticas las columnas y los renglones se comienzan a contar a partir de 1, en C/C++ la cuenta comienza en cero.

```
//programa que captura una matriz matriz.ccp
#include <iostream.h>

int A[30][40];
int i, j;

main() {
  // Captura de los datos
  for( i=0; i<30; i++ ){
    for( j=0; j<40; j++ ){
      cout << "introduce el valor de A[" <<i+1<<","<<j+1<<"]\n";
      cin >> A[i][j];
    };
  };

  // Desplegar el contenido de la matriz A
  for( i=0; i<30; i++ ){
    for( j=0; j<40; j++ ){
      cout << "A[" << i+1 << "," << j+1 << "]=" << A[i][j];
    };
  };

  return 0;
}
```

Ahora bien, para trabajar con una matriz usando la memoria dinámica, tenemos:

```
//programa que captura una matriz matriz.ccp
#include <iostream.h>

int **A;
int i, j, n, m;

main() {
  cout << "número de renglones: "; cin >> n;
  cout << "número de columnas: ";  cin >> m;

  // alojar memoria
  A = new (int *)[n];  // asumir que sí se alojó bien
  for( i=0; i<n; i++ )
    A[i] = new int[m]; // asumir que sí se alojó bien

  // Captura de los datos
  for( i=0; i<n; i++ ) {
    for( j=0; j<m; j++ ) {
      cout << "introduce el valor de A[" <<i+1<<","<<j+1<<"]\n";
      cin >> A[i][j];
    };
  };

  // Desplegar el contenido de la matriz A
  for( i=0; i<n; i++ ) {
    for( j=0; j<m; j++ ) {
      cout << "A[" << i+1 << "," << j+1 << "]=" << A[i][j];
    };
  };

  // desalojar la memoria
  for( i=0; i<n; i++ )
    delete [] A[i];
  delete [] A;
  return 0;
}
```

También es posible trabajar con arreglos tridimensionales, usando memoria estática o bien memoria dinámica.

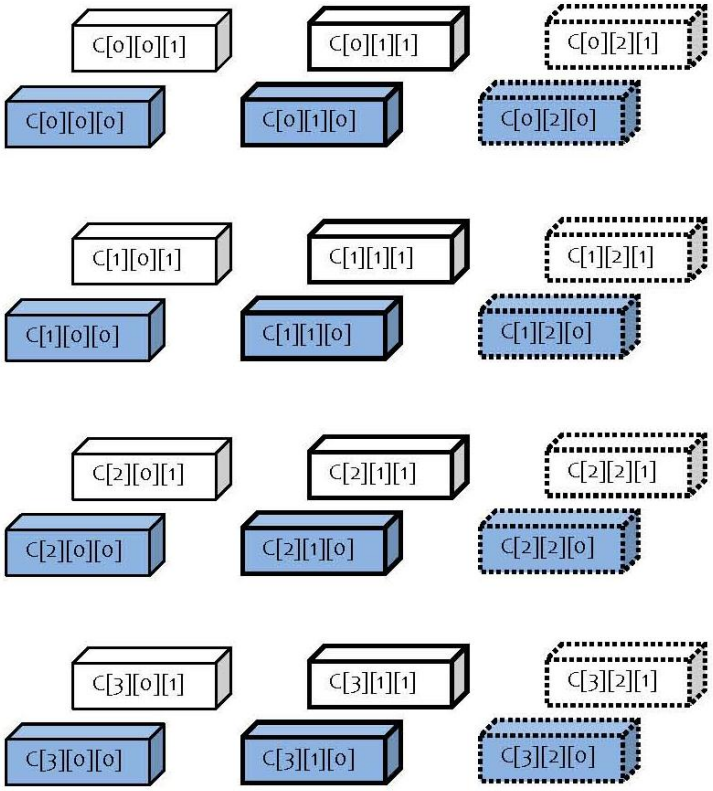


Figura IV-11 Ejemplo de un arreglo tridimensional

En la Figura IV-11 se ilustra un arreglo tridimensional. Si manejamos memoria estática es necesario indicar desde el principio las dimensiones del arreglo: `float C[4][3][2]`. Para usar memoria dinámica es necesario alojar la memoria para poder usar el arreglo.

Ejemplo 4.13.- Hacer un programa que capture los datos de un arreglo multidimensional C de tamaño 2x5x3 y que, una vez capturados, los despliegue en la pantalla.

- Usando memoria estática.
- Usando memoria dinámica.

Solución con memoria estática:

```
//programa que captura un arreglo de 3 dimensiones Dim3.ccp
#include <iostream.h>

main() {
  int C[2][5][3];
  int i, j, k;

  // Captura de los datos
  for( i=0; i<2; i++ )   // para la primera dimensión
  for( j=0; j<5; j++ )   // para la segunda dimensión
  for( k=0; k<3; k++ ) { // para la tercera dimensión
    cout << "C[" << i << ", " << j << << ", " << k <<"]=?";
    cin >> C[i][j][k];
  };
  return 0;
};
```

Solución con memoria dinámica:

```
main() {
  int ***A;
  int n, m, l;

  cout << "De cuantos renglones será la matriz?";
  cin >> n;

  //  Alojar arreglo de apuntadores a int
  A = new int **[n];

  cout << "De cuantas columnas será la matriz?";
  cin >> m;

  // Alojar un arreglo de m columnas para cada renglón
  for( int i=0; i<n; i++ )
    A[i]= new int *[m];

  cout << " de que tamaño es la tercera dimensión? ";
  cin >> l;

  // Alojar en cada renglón y columna otro arreglo de tamaño l
  for( int i=0; i<n; i++ )
    for (int j=0; j < m; j++ )
      A[i][j] = new int[l];

  // Capturar los datos de la matriz
  LeerMatriz(A, n, m, l);

  // Desplegar los datos de la matriz
  ImprimirMatriz(A, n, m, l);

  // Desalojar la memoria
  if( A != NULL ){
    for( int i=0; i<n; i++ ) {
      if( A[i] != NULL ){
        for( int j=0; j<m; j++ )
          if( A[i][j] != NULL )
            delete [] A[i][j];
        delete [] A[i];
      };
    };
    delete [] A;
  };

  return 0;
}
```

Las funciones para leer los datos y desplegarlos son:

```
void ImprimirMatriz( int ***A, int n, int m, int l ) {
  cout << "Los datos de la matriz fueron: \n";
  for( int i = 0; i < n; i++ ) { // para cada fila
    for( int j = 0; j < m; j++ ) { // para cada columna
      for( int k = 0; k < l; k++ ) { // para la profundidad
        cout << "A[" << i << ", " << j << ", "<< k << "] = ";
        cout << A[i][j][k] << endl;
      };
    };
  };
}

void LeerMatriz( int ***A, int n, int m, int l ){
  for( int i = 0; i < n; i++ ) { // para cada fila
    for( int j = 0; j < m; j++ ) { // para cada columna
      for( int k = 0; k < l; k++ ) { // para la profundidad
        cout << "A[" << i << ", " << j <<", "<< k << "] = ?";
        cin >> A[i][j][k];
      };
    };
  };
}
```

IV.3.2 Ejercicios con arreglos dinámicos

Ejercicio 4.7.- Hacer un programa que lea desde el teclado los datos `a(i,j)` de una matriz **`A`** (de tamaño *n* x *m*) y que los despliegue en pantalla. Desalojar la memoria al final.

$$A = \begin{bmatrix} a_{00} & \dots & a_{0j} & a_{0n} \\ \dots & \dots & \dots & \dots \\ a_{i0} & \dots & a_{ij} & a_{in} \\ a_{n0} & \dots & a_{nj} & a_{nn} \end{bmatrix}$$

Solución parcial:

```
// Programa que lee datos de matriz en memoria dinámica (DevCPP)
#include <cstdlib>
#include <iostream>

using namespace std;

int main( int argc, char *argv[] ) {
  int **A;
  int n, m;

  cout << "De cuantos renglones sera la matriz?";
  cin >> n;

  //  Alojar arreglo de apuntadores a int

  cout << "De cuantas columnas sera la matriz?";
  cin >> m;

  // Alojar un arreglo de m columnas para cada renglon


  // Capturar los datos de la matriz
  for( int i = 0; i < n; i++ ){ // para cada fila
    for( int j = 0; j < m; j++ ){ // para cada columna

    };
    // En este ciclo se capturaron las columnas de la fila i
  };

  // Desplegar los datos de la matriz
  cout << "Los datos de la matriz fueron: \n";

  // Desalojar la memoria


  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Solución completa:

```
// Programa que lee datos de matriz en memoria dinámica (DevCPP)
#include <cstdlib>
#include <iostream>

using namespace std;

int main( int argc, char *argv[] ) {
  int **A;
  int n, m;

  cout << "De cuantos renglones sera la matriz?";
  cin >> n;

  //  Alojar arreglo de apuntadores a int
  A = new int *[n];

  cout << "De cuantas columnas sera la matriz?";
  cin >> m;

  // Alojar un arreglo de m columnas para cada renglon
  for( int i=0; i<n; i++ )
    A[i]= new int[m];

  // Capturar los datos de la matriz
  for( int i = 0; i < n; i++ ){ // para cada fila
    for( int j = 0; j < m; j++ ){ // para cada columna
      cout << "A[" << i << ", " << j << "] = ?";
      cin >> A[i][j];
    };
    // En este ciclo se capturaron las columnas de la fila i
  };

  // Desplegar los datos de la matriz
  cout << "Los datos de la matriz fueron: \n";
  for( int i = 0; i < n; i++ ){ // para cada fila
    for( int j = 0; j < m; j++ ){ // para cada columna
      cout << "A[" << i << ", " << j << "] = ?";
      cout << A[i][j] << endl;
    };
    // En este ciclo se capturaron las columnas de la fila i
  };

  // Desalojar la memoria
  for( int i=0; i<n; i++ ){
    if( A[i] != NULL )
      delete [] A[i];
  };
  delete [] A;

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Ejercicio 4.8.- Modificar el programa anterior para que el programa principal haga un llamado a las funciones `LeerMatriz()` para pedir los datos, e `ImprimirMatriz()` para imprimirlos en pantalla. En el programa principal se solicita al usuario el número de renglones *n* y de columnas *m* de la matriz y se aloja la memoria dinámica. Liberar la memoria al final.

Pasar como parámetros la matriz `A`, el número de renglones y de columnas.

Solución:

```
#include <cstdlib>
#include <iostream>
using namespace std;

void ImprimirMatriz( int **A, int n, int m ) {
  cout << "Los datos de la matriz fueron: \n";
  for( int i = 0; i < n; i++ ){ // para cada fila
    for( int j = 0; j < m; j++ ){ // para cada columna
      cout << "A[" << i << ", " << j << "] = ?";
      cout << A[i][j] << endl;
    };
    // En este ciclo se capturaron las columnas de la fila i
  };
}

void LeerMatriz( int **A, int n, int m ) {
  for( int i = 0; i < n; i++ ){ // para cada fila
    for( int j = 0; j < m; j++ ){ // para cada columna
      cout << "A[" << i << ", " << j << "] = ?";
      cin >> A[i][j];
    };
    // En este ciclo se capturaron las columnas de la fila i
  };
}

int main( int argc, char *argv[] ) {
  int **A;
  int n,m;

  cout << "De cuantos renglones sera la matriz?";
  cin >> n;

  //  Alojar arreglo de apuntadores a int
  A = new int *[n];

  cout << "De cuantas columnas sera la matriz?";
  cin >> m;

  // Alojar un arreglo de m columnas para cada renglon
  for( int i=0; i<n; i++ )
    A[i]= new int[m];

  // Capturar los datos de la matriz
  leermatriz(A, n, m);

  // Desplegar los datos de la matriz
  imprimirmatriz(A, n, m);

  // Desalojar la memoria
  for( int i=0; i<n; i++ ){
    if( A[i] != NULL )
      delete [] A[i];
  };
  delete [] A;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Ejercicio 4.9.- Completa el siguiente programa que calcula el producto punto de dos vectores de tamaño *n*, el cual está definido por:

$$\mathbf{u}\cdot\mathbf{v}=\begin{bmatrix}u_1 & u_2 & \ldots & u_n\end{bmatrix}\cdot\begin{bmatrix}v_1\\ v_2\\ \ldots\\ v_n\end{bmatrix}=\sum_{i=1}^{n}u_i v_i$$

```
// Programa que captura el producto punto de dos vectores (DevCPP)
#include <cstdlib>
#include <iostream>

using namespace std;

void PideDatosVector( . . ., . . .){
  for( int i=0; i<tamanio; i++ ){
    cout << "dato "<< i+1 << "=? ";
    cin >> vector[i];
  };
}

double productoPunto( . . ., . . ., int n){
  double prod=0;
  for( int i=0; i<n; i++ )
    . . .
  return . . .;
}

int main( int argc, char *argv[] ) {
  int *v, *u, n;

  cout << "De que tamaño son los vectores U y V?";
  cin >> n;

  // Alojamiento de los vectores


  cout << " ¿Cuales son los datos del vector u? \n";
  PideDatosVector( u, n);
  cout << " ¿Cuales son los datos del vector v? \n";
  PideDatosVector( . . ., . . . );

  cout << "el producto punto de u por v es:"
       << productoPunto(u, v, n);

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Solución:

```
// Programa que captura el producto punto de dos vectores (DevCPP)
#include <cstdlib>
#include <iostream>

using namespace std;

void PideDatosVector( int *vector, int tamanio ) {
  for( int i=0; i<tamanio; i++ ){
    cout << "dato "<< i+1 << "=? ";
    cin >> vector[i];
  };
}

double productoPunto( int *u, int *v, int n) {
  double prod=0;
  for( int i=0; i<n; i++ )
    prod = prod + u[i]*v[i];
  return prod;
}

int main(int argc, char *argv[]) {
  int *v, *u, n;

  cout << "De que tamaño son los vectores U y V?";
  cin >> n;

  // Alojamiento de los vectores

  u = new int [n];
  v = new int [n];

  cout << " ¿Cuales son los datos del vector u? \n";
  PideDatosVector( u, n);
  cout << " ¿Cuales son los datos del vector v? \n";
  PideDatosVector( v, n);

  cout << "el producto punto de u por v es:"
       << productoPunto(u, v, n);

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Ejercicio 4.10.- Codificar las funciones del siguiente programa que calcula el producto interno de una matriz A de tamaño *n* x *m* por un vector de tamaño *n,* el cual está dado por:

$$A \cdot \mathbf{v} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \dots \\ \mathbf{a}_n \end{bmatrix} \cdot \mathbf{v} = \begin{bmatrix} a_1 \cdot \mathbf{v} \\ a_2 \cdot \mathbf{v} \\ \dots \\ a_n \cdot \mathbf{v} \end{bmatrix}$$

```
// Programa que calcula el producto interno de una matriz por
// un vector (DevCPP)

#include <cstdlib>
#include <iostream>

using namespace std;

void PideDatosVector( int *vector, int tamanio ) {
 ...
}

void PideDatosMatriz( int ..., int n, int m ) {
   . . .
}

void prodMatrizVector(. . ., . . ., . . ., . . ., . . .) {
  for( int i=0; i<n; i++ ){
    prod[i] = 0;
    for( int j=0; j<m; j++ )
      . . .
  };
}

int main( int argc, char *argv[] ) {
  int *v, n, m;
  cout << "¿De que tamaño es el vector?"; cin>>n;
  // Alojar el vector v
  v = new int [n];
  cout << " ¿Cuales son los datos del vector v? \n";
  PideDatosVector( v, n);
  int *prodMatrVect, **A;

  // Alojar prodMatrVect y el arreglo de apuntadores a int
  prodMatrVect = new int[n];
  A = new int *[n];

  cout << "La matriz debe tener " << n
       << "renglones. ¿Cuantas columnas tiene?";
  cin >> m;

  // Alojar las columnas de A
  for( int i=0; i<n; i++ )
    A[i] = new int [m];

  PideDatosMatriz( A, n, m);
  prodMatrizVector( A, v, prodMatrVect, n, m);

  cout << "El producto de A por v es:\n ";

  // Desplegar prodMatrVect
  for( int i=0; i<n; i++ )
    cout << prodMatrVect[i] << endl;

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Solución. El código de las funciones es el siguiente.

```
void PideDatosVector( int *vector, int tamanio ) {
  for( int i=0; i<tamanio; i++ ){
    cout << "dato "<< i+1 << "=? ";
    cin >> vector[i];
  };
}
void PideDatosMatriz( int **A, int n, int m){
  for( int i=0; i< n; i++ ) {
    for (int j=0; j< m; j++ ) {
      cout << "a("<< i+1 <<", " << j+1<< " )=? ";
      cin >> A[i][j];
    };
  };
}

void prodMatrizVector( int **A, int *v, int *prod, int n, int m ){
  for( int i=0; i<n; i++ ) {
    prod[i] = 0;
    for( int j=0; j<m; j++ )
      prod[i] = prod[i] + A[i][j]*v[j];
  };
}
```

Ejercicio 4.11.- El producto interno de una matriz ***A*** de *n* por *m* por una matriz ***B*** de *m* por *r* se define como:

$$A\cdot B=\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}\cdot\begin{bmatrix} b_{11} & b_{12} & \dots & b_{1r} \\ b_{21} & b_{22} & \dots & b_{2r} \\ \dots & \dots & \dots & \dots \\ b_{m1} & b_{m2} & \dots & b_{mr} \end{bmatrix}=\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \dots \\ \mathbf{a}_n \end{bmatrix}\cdot\begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \dots & \mathbf{b}_r \end{bmatrix}=\begin{bmatrix} \mathbf{a}_1\cdot\mathbf{b}_1 & \mathbf{a}_1\cdot\mathbf{b}_2 & \dots & \mathbf{a}_1\cdot\mathbf{b}_r \\ \mathbf{a}_2\cdot\mathbf{b}_1 & \mathbf{a}_2\cdot\mathbf{b}_2 & \dots & \mathbf{a}_2\cdot\mathbf{b}_r \\ \dots & \dots & \dots & \dots \\ \mathbf{a}_n\cdot\mathbf{b}_1 & \mathbf{a}_n\cdot\mathbf{b}_2 & \dots & \mathbf{a}_n\cdot\mathbf{b}_r \end{bmatrix}$$

En otras palabras, el resultado es una matriz de *n* por *r* formada con los productos punto de cada renglón de ***A*** (vistos como vectores $\mathbf{a}_i$) por cada columna de ***B*** ($\mathbf{b}_j$). El código en C++ para realizar el producto de dos matrices usando memoria estática es el siguiente:

```
// Código que calcula el producto interno de una matriz por
// otra matriz usando memoria estática

double **prod;
prod = new (double *)[n];
for( int i=0; i<n; i++ )
  prod[i] = new double[r];
for( int i=0; i<n; i++ )
  for( int j=0; j<r; j++ ) {
    prod[i][j] = 0;
    for( int k=0; k<m; k++ )
      prod[i][j] = prod[i][j] + A[i][k]*B[k][j];
  }
cout << "A.B = " << endl;
for( int i=0; i<n; i++ ) {
  for( int j=0; j<r; j++ )
    cout << prod[i][j] << " ";
  cout << endl;
}
```

Hacer un programa que calcule el producto de dos matrices haciendo uso de la memoria dinámica.

Solución parcial:

```
// Programa que calcula el producto interno de dos matrices
// haciendo uso de la memoria dinámica
#include <cstdlib>
#include <iostream>

main() {
  int **A;
  int  n1, m1; // número de renglones y columnas de la matriz A

  cout << "¿De que tamaño es la matriz A?" << endl
       << "renglones =? "; cin>>n1;
  cout << " columnas =? "; cin>>m1;

  // Alojar la matriz A
  . . .

  int **B;
  int n2, m2; // número de renglones y columnas de la matriz B

  // pedir los datos de la matriz B
  cout << "¿De que tamaño es la matriz B?" << endl
       << "renglones =? "; cin>>n2;
  cout << " columnas =? "; cin>>m2;

  if( m1 != n2){
    cout << " Error! el número de columnas de A debe"
         << " ser igual al de renglones de B";
    system("PAUSE");
    return -1;
  };
  // Alojar la matriz B
  . . .

  // Alojar la matriz resultado: prodMatrMatr
  int **prodMatrMatr;
  . . .

  cout << "proporciona los datos de la matriz A \n";
  PideDatosMatriz( A, n1, m1);
  cout << "proporciona los datos de la matriz B \n";
  PideDatosMatriz( B, n2, m2);
  MatrizPorMatriz( A, B, prodMatrMatr, n1, m1, m2);

  cout << "El producto de A por B es:\n ";
  // Desplegar prodMatrMatr
  . . .
};
```

Solución:

```
// Programa que calcula el producto interno de dos matrices
// haciendo uso de la memoria dinámica

#include <cstdlib>
#include <iostream>

main() {
  int **A;
  int  n1, m1; // número de renglones y columnas de la matriz A

  cout << "¿De que tamaño es la matriz A?" << endl
       << "renglones =? "; cin>>n1;
  cout << " columnas =? "; cin>>m1;

  // Alojar la matriz A
  A = new int* [n1];
  for( int i=0; i<n1; i++ )
    A[i]= new int[m1];

  int **B;
  int n2, m2; // número de renglones y columnas de la matriz B

  // pedir los datos de la matriz B
  cout << "¿De que tamaño es la matriz B?" << endl
       << "renglones =? "; cin>>n2;
  cout << " columnas =? "; cin>>m2;

  if( m1 != n2){
    cout << " Error! el número de columnas de A debe"
         << " ser igual al de renglones de B";
    system("PAUSE");
    return -1;
  };

  // Alojar la matriz B
  B = new (int *[n2]);
  for( int i=0; i<n2; i++ )
    B[i]= new int[m2];

  // Alojar la matriz resultado: prodMatrMatr
  int **prodMatrMatr;
  prodMatrMatr = new int *[n1];
  for( int i=0; i<n1; i++ )
    prodMatrMatr[i] = new int[m2];

  cout << "proporciona los datos de la matriz A \n";
  PideDatosMatriz( A, n1, m1);
  cout << "proporciona los datos de la matriz B \n";
  PideDatosMatriz( B, n2, m2);
  MatrizPorMatriz( A, B, prodMatrMatr, n1, m1, m2);

  cout << "El producto de A por B es:\n ";
  // Desplegar prodMatrMatr
  for( int i=0; i<n1; i++ ) {
    for( int j=0; j<m2; j++ ){
      cout << prodMatrMatr[i][j] << " ";
    };
```

```
  };

  //Desalojar memoria
  for( int i=0; i<n1; i++ )
    delete A[i];
  delete A;

  for( int i=0; i<n2; i++ )
    delete B[i];
  delete B;

  return 0;
}
```

¿Cómo codificarías las funciones `PideDatosMatriz` y `MatrizPorMatriz`?

Solución:

```
void PideDatosMatriz( int **Matriz, int n, int m ) {
  for( int i=0; i< n; i++ ) {
    for( int j=0; j< m; j++ ) {
      cout << "("<< i+1 <<", " << j+1<< " )=? ";
      cin >> Matriz[i][j];
    };
  };
}

void MatrizPorMatriz( int **A, int **B, int **prodMatrMatr,
                      int n1,int n2,int m2 ) {
  for( int i=0; i<n1; i++ ){
    for( int j=0; j<m2; j++ ){
      prodMatrMatr[i][j] = 0;
      for( int k=0; k<n2; k++ ){
         prodMatrMatr[i][j]= prodMatrMatr[i][j] + A[i][k]*B[k][j];
      };
    };
  };
}
```

Capítulo V Fundamentos de diseño modular

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

Comprender los conceptos y elaborar programas que contengan:

- Funciones
- Subrutinas

Introducción

En la práctica, los programás que se desarrollan son mucho más grandes que los programás que se ven en un salón de clases o en un libro de programación como éste. De hecho, los programas comerciales tienen del orden de 100 mil líneas de código en proyectos de mediano tamaño. De aquí la necesidad de llevar a cabo una codificación ordenada y lo más clara posible.

Existe un principio de diseño llamado *divide y vencerás* que consiste en construir un programa a partir de partes o componentes más pequeños que resultan más fáciles de desarrollar y de darles mantenimiento. Para esto, es necesario definir qué parte del programa será puesto en qué componente. Lo más fácil es separar los componentes de manera que cada uno lleve a cabo alguna función específica.

En este capítulo introducimos el concepto de función y de subrutina los cuales pueden ser especificados en casi cualquier lenguaje de programación. Existen otras técnicas más avanzadas para crear módulos de software sin embargo todas tienen como fundamento el uso de subrutinas y funciones, así que resulta fundamental el conocerlas a fondo.

V.1 Subrutinas

Las subrutinas son propiamente subprogramas que realizan alguna tarea, normalmente solo una pequeña parte de las tareas que debe realizar un programa completo. Las subrutinas se echan a andar desde el programa principal o desde alguna otra subrutina las veces que sea necesario. De esta forma se le quita responsabilidad (sobre una tarea específica) al programa que las llama y se le transfiere dicha responsabilidad a la subrutina. Esto tiene consecuencias muy favorables para poder realizar programas grandes. La primera es que es mucho más sencillo entender bien qué hace cada subprograma o subrutina lo que permite a los programadores revisar que todo esté bien en su lógica. Así, es posible entonces ir creando un programa por partes que poco a poco se van terminando y finalmente integrando en uno solo que realiza todas las tareas deseadas. Esto permite, al mismo tiempo, una mejor organización cuando los programas los realiza un equipo de personas. Cuando un programador escribe una subrutina, los demás pueden usarla para hacer sus propios programas sabiendo que esa tarea ya está implementada en esa subrutina, ahorrándose el tiempo de realizarla él mismo y evitándose la duplicación de código. A esto se le conoce como el principio de reutilización de código.

La ejecución de subrutinas es como sigue: cuando un programa hace un llamado (invoca) a una subrutina, éste interrumpe su ejecución y espera a que la subrutina termine y luego continúa su ejecución a partir de la siguiente instrucción al llamado. Ver Figura V-1.

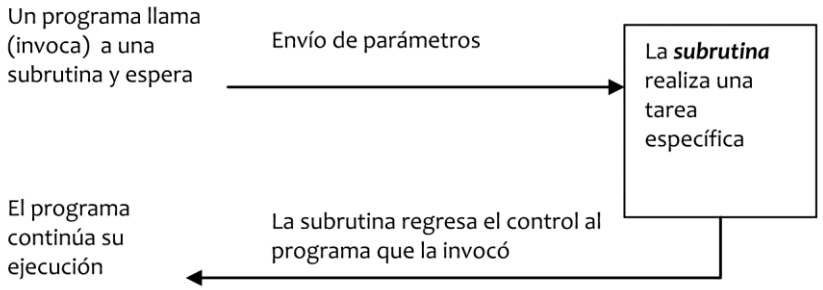


Figura V-1 *Llamado a una Subrutina.*

El siguiente programa es un ejemplo muy simple de un programa principal que ilustra cómo se invocan tres subrutinas. La primera es una subrutina dedicada a solicitar al usuario los datos de entrada que necesita el programa para producir un resultado. La segunda lleva a cabo todos los cálculos para producir los resultados y los guarda en memoria. La tercera se dedica a mostrar en pantalla los resultados en un formato legible para el usuario.

```
#include<iostream.h>

main() {
  PideDatos(); // esta función pide los datos de entrada
  HacerCalculos(); // pone en variables los resultados
  GuardaResultados(); // guarda los resultados en disco

  return 0;
}
```

Como puede observarse, el programa principal es muy corto y esto lo hace fácil de leer y, por lo tanto, es fácil entender la estructura principal del programa. Esta simple ventaja, en programas reales o prácticos, es muy grande. Es altamente recomendable que, cuando se está desarrollando un programa y se nota que el programa ha crecido considerablemente, se piense en reescribirlo separando sus tareas en subrutinas.

V.2 Funciones

Las funciones en programación son simplemente subrutinas que producen un resultado dado por un valor correspondiente a un cierto tipo de dato predefinido. Este valor es enviado desde la función hacia el programa que la invocó al momento de regresarle el control. A este valor se le llama el valor de regreso de la función que normalmente depende del valor de los parámetros actuales que recibe al ser invocada. Es por esto que la invocación de funciones se hace desde una expresión que requiere ser evaluada. En la Figura V-2 se ilustra esto gráficamente.

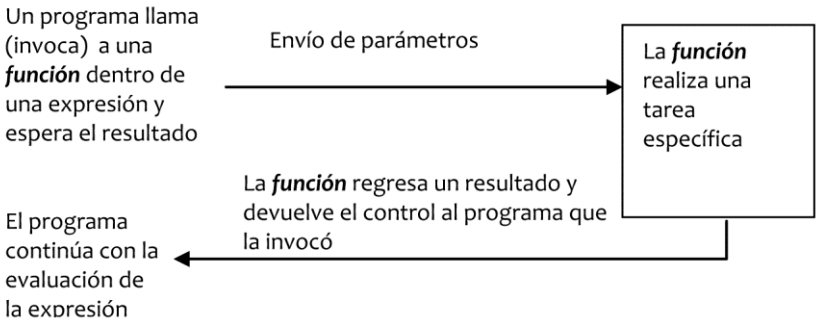


Figura V-2 *Llamado a una Función.*

Se considera una buena práctica que cada función se limite a efectuar una sola tarea bien definida y que el nombre de la función exprese la tarea que ejecuta.

V.3 Biliotecas de funciones

Las funciones pueden construirse desde cero, o bien pueden utilizarse funciones "pre-fabricadas" que ya se encuentran dentro de alguna biblioteca del compilador C/C++.

La biblioteca estándar de C/C++ contiene un amplio conjunto de funciones para realizar: cálculos matemáticos comunes, manipulaciones de cadenas de caracteres, operaciones dentrada/salida, comprobación de errores y muchas otras aplicaciones útiles.

El uso de estas funciones simplifica el trabajo del programador, sin embargo, no es posible modificarlas, solo se les invoca.

Para poder invocar a estas funciones es necesario incluir el archivo de encabezado (header) que contenga la declaración del *prototipo* de la función requerida (ejemplo: math.h). Véase en los programas que se han visto hasta este capítulo que frecuentemente se usan dos bibliotecas: **`iostream.h`** y **`math.h`**.

b) Las funciones definidas por el programador:

Son las que éste escribe para definir ciertas tareas específicas que podrán utilizarse en varios puntos del programa.

Es responsabilidad del programador el que funcionen bien.

Son más versátiles ya que se les puede modificar.

V.3.1 Funciones de la biblioteca matemática

Las funciones de la biblioteca matemática le permiten al programador efectuar ciertos cálculos matemáticos comunes. Las funciones normalmente se invocan escribiendo el nombre de la función seguido de un paréntesis izquierdo, del argumento (o lista de argumentos separados por comás) de la función y de un paréntesis derecho. Por ejemplo, un programador que desee calcular y mostrar la raíz cuadrada de 900.0 puede escribir:

```
cout << sqrt(900.00);
```

Para utilizar las funciones de la biblioteca matemática, es necesario incluir el archivo de encabezado **math.h.** Los argumentos de una función pueden ser constantes, variables o expresiones. Por ejemplo:

Si c1 = 13.0, d=3.0 y f = 4.0, entonces la instrucción: `cout<< sqrt (c1 + d*f);` calcula y muestra la raíz cuadrada de 13.0 + 3.0*4.0 = 25.0, es decir 5.

Todas las funciones de la biblioteca matemática mostradas en la *Tabla 5.1*, excepto la función *`abs()`*, admiten como argumento tanto número enteros como reales. Sin embargo, el resultado que arrojan es un `float`, con excepción de la función *`abs()`* cuyo resultado es un entero. A continuación se muestra una tabla con las funciones matemáticas más comunes.

Invocación	Descripción	Ejemplo
ceil(x)	Redondea x al entero más pequeño no menor que x	ceil(9.2) es 10.0 ceil(-9.8) es -9.0
cos(x)	Coseno trigonométrico de x (en radianes)	cos(1.5707) es 0.0
exp(x)	Función exponencial e^x	exp(1) es 2.71828
fabs(x)	Valor absoluto de x (reales)	si x > 0 → abs(x) es x si x = 0 → abs(x) es 0 si x < 0 → abs(x) es -x
abs(x)	Valor absoluto de x (enteros)	si x > 0 → abs(x) es x si x = 0 → abs(x) es 0 si x < 0 → abs(x) es -x
floor(x)	Redondea x al entero más grande no mayor que x	floor(9.2) es 9.0 floor(-9.8) es -10.0
log(x)	Logaritmo natural de x (base e)	log(2.718282) es 1.0
log10(x)	Logaritmo base 10 de x	log10(100) es 2.0
sqrt(x)	Raíz cuadrada de x	sqrt(25.0) es 5
pow(x, y)	x elevado a la potencia y	pow(2, 5) es 32 pow(16,0.5) es 4
sin(x)	Seno trigonométrico de x (x en radianes)	sin(1.5707) es 1.0
tan(x)	Tangente trigonométrica de x (x en radianes)	tan(0.78) es 1.0

Tabla 5.1 Las funciones matemáticas más comunes.

V.3.1.1 Ejercicios

Ejercicio 5.1.- Obtener el resultado de cada una de las siguientes funciones indicando con el punto decimal cuando el resultado sea un `float` y sin punto decimal cuando es un `int`.

```
sqrt (16.0) =
pow(2,4) =
pow(2.0,4) =
pow(1.1,2) =
abs (3) =
```

```
abs (-3) =
abs (0.7) =
fabs (-3.0) =
fabs (-3.5) =
pow(3.0,2)/2.0=
7/abs(-2) =
```

Solución:

```
sqrt (16.0) =4
pow(2,4) =16.0
pow(2.0,4) =16.0
pow(1.1,2) =1.21
abs (3) =3
abs (-3) =3
abs (0) =0
fabs (-3.0) =3.0
fabs (-3.5) =3.5
pow(3.0,2)/2.0=4.5
7/abs(-2) =3
```

Nótese que al dividir un entero entre otro entero el resultado es otro entero, por lo tanto se trunca la parte decimal de la división.

Ejercicio 5.2.- Convertir las siguientes expresiones algebraicas a lenguaje C/C++.

$\sqrt{x+y}$ =

x^{y+7} =

$\sqrt{area + margen}$ =

$\sqrt{\frac{area + margen}{año}}$ =

|x-y | =

Solución:

$\sqrt{x+y}$ = `sqrt(x+y)`

x^{y+7} = `pow(x,(y+7))`

$\sqrt{area + margen}$ = `sqrt(area + margen)`

$\sqrt{\frac{area + margen}{año}}$ = `sqrt((tiempo+ marca)/año)`

|x-y| = `abs(x-y)`

V.4 Funciones y subrutinas definidas por el programador

Además de las funciones existentes en las bibliotecas, es posible que el programador defina sus propias funciones y subrutinas. En esta sección se verá cómo puede hacerse esto en C y C++.

V.4.1 Partes de una función

Para que el programador pueda crear una función es necesario primero *declararla* y después *definirla*.

Declaración de una función: Para declarar una función se deben especificar sus características externas:

- El nombre de la función.
- El tipo de resultado que proporciona como valor de retorno.
- La lista de parámetros o argumentos sobre los que va a operar y el tipo de cada uno.

Ejemplo: nombre `f`, resultado tipo `double`, parámetros `x` y `y` de tipo `double`.

Definición de la función.- Una vez dado lo anterior se debe especificar el procedimiento (un programa) que debe usarse para transformar los parámetros en el resultado.

Ejemplo: $f(x,y) = x^2 + 2y^2 - xy$

En la siguiente sección se explica y se dan ejemplos de la construcción de funciones.

V.4.2 Declaración y Definición de las funciones

La *declaración de una función ó Prototipo* da las características externas de la función pero no las internas.

Partes de la *declaración* de una función.- En la declaración de una función se indica cuales son los argumentos ó *parámetros* que recibe y el tipo de cada uno de ellos. También se indica el tipo del valor que regresa como resultado. A esta declaración de la función se le llama *prototipo* de la función. La sintaxis en C y C++ para el prototipo es la siguiente:

```
<tipo_resultado> <nombre_funcion>( <lista de parámetros> );
```

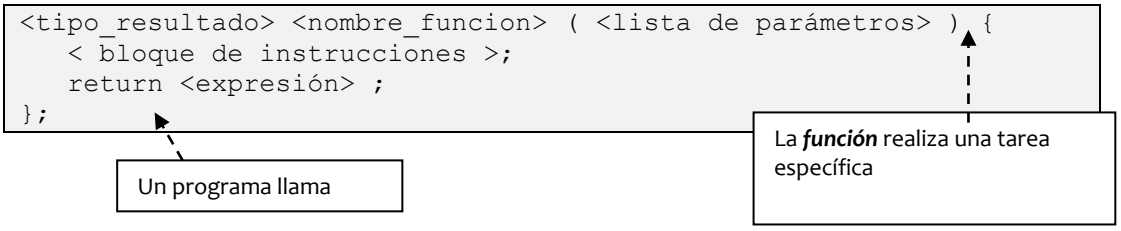
Ejemplos de prototipo de algunas funciones.

```
float area_triangulo(float base, float altura);
int sumatoria( int arreglo[], int tamaño);
double promedio( int arreglo[], int tamaño);
long int  factorial( int numero);
void captura_datos( void );
```

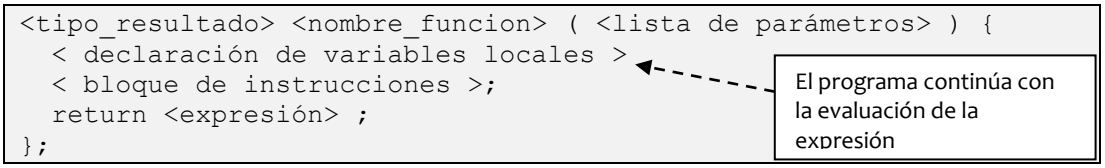
La palabra `void` se usa para indicar que una función no entrega (o devuelve) un valor ó que no recibe parámetros cuando se le invoca.

Como puede observarse, la palabra `void` sirve para construir *subrutinas* (también llamadas procedimientos) en lenguaje C/C++, cuando se le usa como tipo de resultado de una función.

Partes de la *definición* de una función.- La *definición de una función* consta de un "*encabezado*" de la función y del "*cuerpo*" de la función encerrado entre llaves:



Las variables declaradas en el cuerpo de la función son *variables locales* y por definición solamente son accesibles dentro del mismo. Estas se crean cuando se ejecuta la función y se destruyen cuando finaliza la ejecución de la función.



El *tipo del resultado* especifica qué tipo de datos retorna la función. Éste puede ser cualquier tipo fundamental o un tipo definido por el usuario, pero no puede ser un arreglo o una función. Si no se especifica, por default, la función devuelve un

entero. Por medio de la instrucción `return` se devuelve el resultado de una función al punto desde el cual se invocó.

```
<tipo_resultado>  <nombre_funcion> (<lista de parámetros>) {
  < declaración de variables locales >
  < bloque de instrucciones >;
  return <expresión> ;
};
```

El *encabezado* es como el prototipo pero

La instrucción `return( )` puede ser o no la última y puede aparecer más de una vez en el cuerpo de la función. En el caso de que la función no regrese un valor, se puede omitir o especificar simplemente un `return`. Por ejemplo:

```
void tipo_resultado  nombre_funcion (<lista de parámetros>) {
     < declaración de variables locales >
     < bloque de instrucciones >;
      return;
};
```

Cuerpo de la función

Ejemplos de definición de funciones:

```
float area_triangulo( float base, float altura ) {
  return base * altura / 2 ;
};
```

```
int sumatoria( int arreglo[], int tamaño) {
  int suma = 0;

  for( int i = 0; i < tamaño; i++ )
    suma = suma + arreglo[i];

  return suma;
};
```

Nota importante: En C y C++ no es posible definir una función dentro de otra como en algunos otros lenguajes de programación.

V.4.3 Contexto de definición de variables

Las variables pueden declararse en diferentes lugares del programa y, dependiendo del lugar en el que éstas se declaren tienen validez general o solo en ciertos lugares. En este contexto, se distinguen dos tipos de variables.

Las variables **globales**: Son las que se declaran "afuera" de las funciones y del `main`. Se pueden usar en cualquier función que esté definida después de la declaración de la variable.

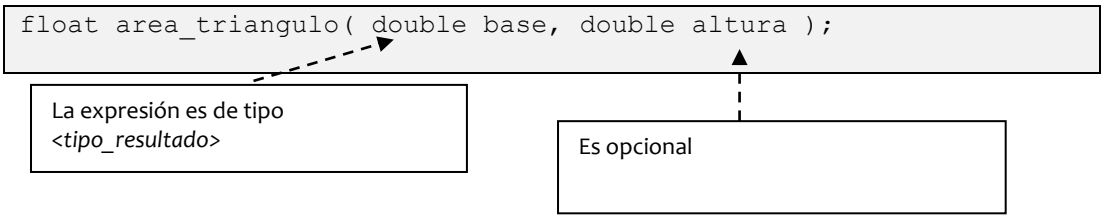
Las variables **locales**: Son las que están declaradas dentro de alguna función incluyendo a la función `main`. Solamente pueden usarse dentro de la función y a partir de su declaración.

En C++ la variable local puede tener efecto solamente en un fragmento de código, por ejemplo, en un `for` puede declararse el contador `i` como una variable local: `for( int i=0; i<10; i++ ) cout << i;`

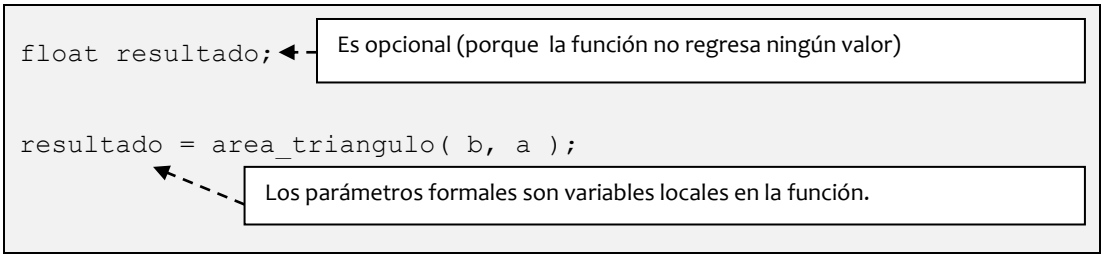
En este caso la variable `i` solo es "visible" dentro del ciclo `for`. Nota: hay que tener cuidado con este tipo de declaraciones, ya que algunos compiladores asumen una declaración dentro del `for` como una declaración para toda la función y ya no permiten volver a declararla nuevamente dentro de la mísma en otro for subsiguiente.

V.4.4 Los parámetros

Los parámetros vistos desde la función se llaman *parámetros formales*. Los *parámetros formales* de una función son las variables que reciben los valores de los argumentos especificados en la llamada a la función. Consisten en una lista de identificadores con sus tipos, separados por comás. Éstos deben estar descritos en el prototipo y en la definición de la función.



Los parámetros vistos desde el llamado a la función se llaman *parámetros actuales*. Para que una función se ejecute hay que *llamarla* (o *invocarla*). La llamada a una función consta del nombre de la misma y de una lista de argumentos, denominados *parámetros actuales* encerrados entre paréntesis y separados por comás, por ejemplo, una llamada a la función del ejemplo anterior puede ser:



```
Llamado a la función ( <parámetros actuales> )
                          se requiere especificar el tipo
                          para cada parámetro de la lista.

funcion( <parámetros formales> )
                          resultado debe ser del mismo tipo del valor
                          que regresa la función
```

Si la definición de la función aparece antes de la primera vez que se usa en el programa, entonces no es necesario el prototipo de la función. En este caso, la definición de la función también actúa como prototipo de la función.

Ejemplo 5.1.-

```
// función que obtiene el área de un triángulo  trianArea.cpp
#include<iostream.h>
#include<conio.h>

float area_triangulo( float base, float altura ) {
  return( base * altura / 2 );
};

main() {
  char c;
  float a,b, area;
  do
  {
     cout<< "base del triangulo?";
     cin>> b;
     cout<< "altura?";
     cin>> a;
     area = area_triangulo( b,a );
     cout<< "el area del triangulo es:"<< area;
     cout<<"oprimir una tecla para continuar, ""ESC"" para salir";
     c= getch();
  }while(c!=27);
  return 0;
}
```

Los parámetros actuales son los datos

El valor que devuelve la función se guarda en la variable `resultado`

Los parámetros formales reciben el valor de los parámetros actuales

Ejemplo 5.2.-

```
// aquí se usa el prototipo
#include<iostream.h>

int sumatoria( int arreglo[], int tamaño );

main() {
  int A[5]={0,1,2,3,4}, sumaA;
  int B[4]={35,22,15,17}, sumaB;

  sumaA = sumatoria( A, 5 );
  sumaB = sumatoria( B, 4 );

  cout << "la suma de A es: "<< sumaA << endl;
  cout << "la suma de B es: "<< sumaB;

  return 0;
}

int sumatoria( int arreglo[], int tamaño ) {
  int suma = 0;
  for( int i = 0; i < tamaño; i++ )
    suma = suma + arreglo[i];
  return(suma);
}
```

```
// aquí NO se usa el prototipo
#include<iostream.h>

int sumatoria( int arreglo[], int tamaño ) {
  int suma = 0;
  for( int i = 0; i < tamaño; i++ )
    suma = suma + arreglo[i];
  return(suma);
}

main() {
  int A[5]={0,1,2,3,4}, sumaA;
  int B[4]={35,22,15,17}, sumaB;

  sumaA = sumatoria( A, 5 );
  sumaB = sumatoria( B, 4 );
  cout << "la suma de A es: "<< sumaA << endl;
  cout << "la suma de B es: "<< sumaB;
 return 0;
}
```

V.4.5 Ejercicios de funciones definidas por el programador

Ejercicio 5.3.- Hacer un programa que pida la temperatura en grados Farenheit y que llame a la función:`F_a_Centigrados(…..)` para obtener la temperatura en grados centígrados. (Trabajar con `double`).

Formula:

C = (F – 32)*5 / 9

El parámetro que recibe la función es la temperatura en grados Farenheit.

Solución:

1.- Determinar *entradas y salidas.* La entrada es una temperatura en grados Fahrenheit, a la que llamaremos `Fahrenheit` y será de tipo `double`. La salida será el valor que devuelva la función `F_a_Centigrados` la cual será un `double`.

2.- Especificar operaciones. Se solicita al usuario una temperatura en grado Fahrenheit y se envía como parámetro a la función `F_a_Centigrados` para que regrese la temperatura en grados Centígrados al aplicar la siguiente fórmula:

C = (F – 32)*5 / 9

3.- *Codificación*. A continuación se presenta el programa sin la definición de la función. Como se definiría la función `F_a_Centigrados` ?

Solución parcial:

```
// Programa para conversion de grados
// farenheit --> centigrados
// F_a_C.CPP
#include<iostream.h>

tipo F_a_Centigrados( parametro ) {
  < cuerpo de la función >;
};

main() {
  double farenheit;
  cout << " Programa para convertir grados Farenheit a
           centigrados \n ";
  cout << " Cual es la temperatura en grados Farenheit?";
  cin >> farenheit;

  cout << farenheit << " equivale a: "
       << F_a_Centigrados(farenheit) << " centigrados".

  return 0;
}
```

A continuación se muestra el programa completo:

```
// Programa para conversion de grados
// farenheit --> centigrados
// F_a_C.CPP
#include<iostream.h>

double F_a_Centigrados( double f ) {
  return( ( ( f - 32.0 ) * 5.0 ) / 9.0 );
};

main() {
  double farenheit;
  cout << " Programa para convertir grados Farenheit a
           centigrados \n ";
  cout << " Cual es la temperatura en grados Farenheit?";
  cin >> farenheit;

  cout << farenheit << " equivale a: "
       << F_a_Centigrados(farenheit) << " centigrados".

  return 0;
}
```

Ejercicio 5.4.- Hacer un programa que pida la temperatura en grados Centígrados y que llame a la función: `C_a_Farenheit(…..)` para obtener la temperatura en grados Farenheit. (Trabajar con `double`).

Formula:

$$F = C*9 / 5 + 32$$

El parámetro que recibe la función es la temperatura en grados Centígrados.

Solución.

1.- Determinar *entradas y salidas.* La entrada es una temperatura en grados Centígrados, a la que llamaremos `centigrados` y será de tipo `double`. La salida será el valor que devuelva la función `C_a_Farenheit` la cual será un `double`.

2.- Especificar operaciones. Se solicita al usuario una temperatura en grado Centígrados y se envía como parámetro a la función `C_a_Farenheit` para que regrese la temperatura en grados Centígrados al aplicar la siguiente fórmula:

$$F = C*9 / 5 + 32$$

3.- *Codificación*. A continuación se presenta el programa sin la definición de la función. Como se definiría la función `C_a_Farenheit` ?

Solución parcial:

```
// Programa para conversion de grados
//  centigrados --> Farenheit
// C_a_F.CPP
#include<iostream.h>

tipo C_a_Farenheit( parametro ) {
  < cuerpo de la función >;
};

main() {
  double centigrados;
  cout << " Programa para convertir grados centigrados a
            Farenheit \n ";
  cout << " Cual es la temperatura en grados centigrados?";
  cin >> centigrados;

  cout << centigrados << " equivale a: "
       << C_a_Farenheit( centigrados ) << " Farenheit".

  return 0;
}
```

Solución completa:

```
// Programa para conversion de grados
//  centigrados --> Farenheit
// C_a_F.CPP
#include<iostream.h>
double C_a_Farenheit( double c ) {
  return( 9.0 * c / 5.0 + 32.0 );
};

main() {
  double centigrados;

  cout << " Programa para convertir grados centigrados a
            Farenheit \n ";
  cout << " Cual es la temperatura en grados centigrados?";
  cin >> centigrados;

  cout << centigrados << " equivale a: "
       << C_a_Farenheit( centigrados ) << " Farenheit".

  return 0;
}
```

V.5 Modos de paso de parámetros

Existen tres maneras de pasar parámetros a una función:

1.- *Paso por valor*: se pasa una copia del valor de una variable que no es de tipo arreglo, el contenido original de la variable queda sin modificar, este tipo de paso de parámetros se utiliza cuando los parámetros deben permanecer sin cambios.

2.- *Paso por dirección:* Se pasa un arreglo o un apuntador a una función. Este tipo de llamado se utiliza cuando la función receptora va a cambiar los parámetros que se le envían y esos parámetros deben ser conocidos en la función que llama.

3.- *Paso por referencia:* Se utiliza con el mismo propósito que el "paso por dirección", produce el mismo resultado de cambiar el contenido de los parámetros recibidos, pero la sintaxis es más clara debido a que no se tiene que poner el operador "*" antes de la variable en todos los lugares de la función receptora donde ésta aparece.

Ejemplo 5.3.- A continuación se presenta un programa donde se ejemplifica el paso de parámetros por valor y por dirección.

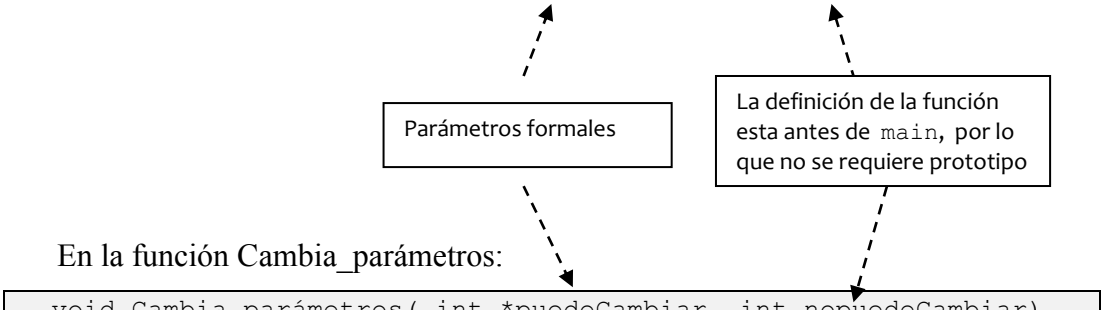
```
// Pasa y recibe parámetros por valor y
// por dirección
// VALOR_DIR.CPP
// (programa de Greg Perry)

#include<iostream.h>
#include<stdlib.h>    //contiene el prototipo para exit()

void  Cambia_parametros( int *puedeCambiar, int nopuedeCambiar );

main() {
  int puedeCambiar = 8;
  int nopuedeCambiar = 16;
  cout<<"Antes de llamar a Cambia_parámetros: \n";
  cout<<"puedeCambiar es:"<< puedeCambiar <<endl;
  cout<<"nopuedeCambiar es:"<< nopuedeCambiar <<endl;

  Cambia_parámetros( &puedeCambiar, nopuedeCambiar );

  cout<<"Después de llamar a Cambia_parámetros:\n";
  cout<<"puedeCambiar es:"<< puedeCambiar <<endl;
  cout<<"nopuedeCambiar es:"<< nopuedeCambiar <<endl;

  return 0;
};

// Recibe dos parámetros uno por dirección y otro por valor
void  Cambia_parametros(int *puedeCambiar, int nopuedeCambiar) {
  *puedeCambiar = 24;   // Lo que apunta la variable
  nopuedeCambiar = 24;

  cout << "Valores locales de ""Cambia_parámetros"" : \n";
  cout << "puedeCambiar es:"   << *puedeCambiar  << endl;
  cout << "nopuedeCambiar es:" << nopuedeCambiar << endl;

  return;
}
```

Cuando se corre este programa, la salida es la siguiente:

```
Antes de llamar a Cambia_parámetros:
puedeCambiar es : 8
nopuedeCambiar es: 16
Valores locales de "Cambia_parametros":
puedeCambiar es : 24
nopuedeCambiar es: 24
Después de llamar a Cambia_parametros:
puedeCambiar es : 24
nopuedeCambiar es: 16
```

En la invocación desde el programa principal, main():

```
Cambia_parámetros( &puedeCambiar ,   nopuedeCambiar);
```



En la función Cambia_parámetros:

```
void Cambia_parámetros( int *puedeCambiar, int nopuedeCambiar)
```

Nota: Todos los arreglos se pasan automáticamente por dirección, ya que el nombre de los arreglos es un apuntador al inicio del arreglo. No se puede pasar un arreglo por valor en C++. Recordar que un apuntador contiene la dirección de los datos.

Ahora veremos un ejemplo del paso de parámetros por referencia, es más conveniente pensar que se "recibe" por referencia. Una función recibe parámetros por referencia mientras la lista de parámetros recibida contenga el signo: `&`.

El paso por referencia es más limpio que el paso por dirección y evita el uso del signo `*` dentro de la función, es decir, el parámetro por referencia se trata como a una variable local normal.

Ejemplo 5.4.- A continuación se presenta un programa donde se ejemplifica el paso de parámetros por dirección y por referencia.

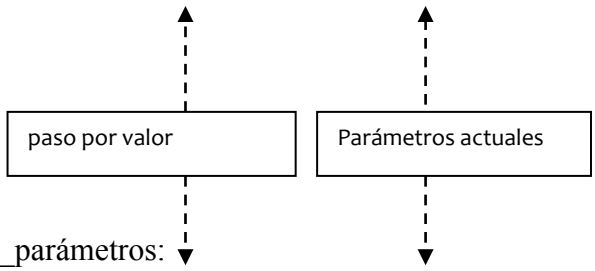
```
// Pasa y recibe parámetros por direccion y por referencia
// DIR_REF.CPP
// (programa de Greg Perry)

#include<iostream.h>
#include<stdlib.h>    //contiene el prototipo para exit()

void Cambia_parametros( int *puedeCambiar1, int &puedeCambiar2 );
                                                  // Nótese  el &

main() {
  int puedeCambiar1 = 8;
  int puedeCambiar2 = 16;
  cout<<"Antes de llamar a Cambia_parámetros: \n";
  cout<<"puedeCambiar1 es:"<< puedeCambiar1 <<endl;
  cout<<"puedeCambiar2 es:"<< puedeCambiar2 <<endl;

  Cambia_parámetros( &puedeCambiar1, puedeCambiar2 );

  //Nótese  que no se usa el & en el parámetro 2
  cout<<"Después de llamar a Cambia_parámetros:\n";
  cout<<"puedeCambiar1 es:"<< puedeCambiar1 <<endl;
  cout<<"puedeCambiar2 es:"<< puedeCambiar2 <<endl;

  return 0;
}

// Recibe dos parámetros uno por dirección y otro por referencia
void Cambia_parametros( int *puedeCambiar1, int  &puedeCambiar2 ){
                                                 // Nótese  el &
  *puedeCambiar1 = 24;// Lo que apunta la variable
  puedeCambiar2 = 24; // Aquí ya no se necesita el *
                      // pero el resultado es el mismo

  cout << "Valores locales de ""Cambia_parámetros"": \n";
  cout << "puedeCambiar1 es:" << *puedeCambiar1 << endl;
  cout << "puedeCambiar2 es:" << puedeCambiar2  << endl;

  return;
}
```

Cuando se corre este programa, la salida es la siguiente:

```
Antes de llamar a Cambia_parámetros:
puedeCambiar1 es:   8
puedeCambiar2 es: 16
Valores locales de la funcion "Cambia_parámetros":
puedeCambiar1 es : 24
puedeCambiar2 es : 24
Después de llamar a Cambia_parámetros:
puedeCambiar1 es: 24
puedeCambiar2 es: 24
```

En la invocación desde el programa principal, main():

```
Cambia_parámetros( &puedeCambiar1, puedeCambiar2 );
```



En la función Cambia_parámetros:

```
void Cambia_parámetros( int *puedeCambiar1, int &puedeCambiar2 )
```

El pasar parámetros por valor añade algunas ineficiencias al programa. Cuando se pasan variables de estructura grandes, C++ tiene que ocupar un tiempo de ejecución en copiar los datos a una variable local.

Si la eficiencia es importante, se usa el paso por referencia paro precediendo el valor recibido con el modificador "const" para que la función receptora no cambie el/los parámetros:

```
// Archivo: REFERENCIA.CPP
//Ejemplo de paso por referencia
// (programa de Greg Perry)
#include<iostream.h>

void doblalo(int &i);

main() {
  int i=20;
  doblalo(i); // Duplica i
  cout<< "i es ahora" << i << "\n";
  return (0);
}

void doblalo(int &i) {
  i =  2*i;
}
```

V.5.1 Ejercicios de paso de parámetros

Ejercicio 5.5.- Hacer un programa que llame a la función `promedio`, la cual regresa como resultado el promedio de todas las cantidades recibidas dentro del arreglo de reales:

```
float  calific[7];
```

Es opcional usar el prototipo y definir la función después del `main` ó definir la función antes del `main` (sin usar el prototipo). Las calificaciones del arreglo se deben pedir al usuario antes de llamar a la función.

Solución:

1.- Determinar *entradas y salidas.* La entrada es un arreglo de calificaciones dadas por el usuario al que llamaremos `calific`, será de tipo `double` y contiene y elementos. La salida será el valor que devuelva la función `promedio` el cual será un `float`.

2.- Especificar operaciones. Después de solicitar al usuario las 7 calificaciones se envía el arreglo `calific` como parámetro a la función `promedio` para que regrese el promedio de las calificaciones contenidas en el arreglo.

3.- *Codificación*. A continuación se presenta el programa sin la definición de la función. Como se definiría la función `promedio` ?

Solución parcial:

```
// Obtener el promedio de un arreglo llamando a una función
//  funPromedio.cpp

#include<iostream.h>

<tipo> promedio( <parametro> ) {
  < cuerpo de la función >
};

main() {
  float calific[7];

  // pedir al usuario las 7 calificaciones
  for( int i = 0; i < 7; i++ ){
    cout << "\n Cual es la calificacion numero " << i+1;
    cin >> calific[i];
  };
  cout << "Tu promedio es:" << promedio( calific );

  return 0;
}
```

Solución completa:

```
// Obtener el promedio de un arreglo llamando a una función
//  funPromedio.cpp

#include<iostream.h>

float  promedio( float arreglo[] ) {
  float suma = 0;
  for( int j = 0; j < 7; j++ )
    suma = suma + arreglo[j];
  return( suma/7 );
};

main() {
  float calific[7];

  // pedir al usuario las 7 calificaciones
  for( int i = 0; i < 7; i++ ){
    cout << "\n Cual es la calificacion numero " << i+1;
    cin >> calific[i];
  };
  cout << "Tu promedio es:" << promedio( calific );

  return 0;
}
```

Ejercicio 5.6.- Modificar el ejercicio anterior haciendo que la función pueda obtener el promedio de dos arreglos de tamaño diferente. El contenido de los arreglos es el siguiente:

```
 float alumno1[3]={9.7, 8.2, 7.0};
 float alumno2[5]={5.9, 10.0, 8.5, 9.1, 6.2};
```

¿Cómo se define la función ahora?

Solución parcial:

```
// Con función que obtiene el promedio de dos arreglos
// de tamaño diferente promArreglos.cpp

#include<iostream.h>

<tipo>  promedio( <parametros> ) {
  < cuerpo de la función >
};

main() {
  float alumno1[3]={9.7, 8.2, 7.0};
  float alumno2[5]={5.9, 10.0, 8.5, 9.1, 6.2};
  cout << "Promedio del alumno 1 es:"
       << promedio( alumno1, 3 );
  cout << "\n Promedio del alumno 2 es:"
       << promedio( alumno2, 5 );

  return 0;
}
```

Solución completa:

Para que la función pueda trabajar con arreglos de tamaño diferente es necesario agregar un parámetro adicional que indique el tamaño del arreglo, de esta forma tenemos la solución siguiente:

```
// Con función que obtiene el promedio de dos arreglos
// de tamaño diferente  promArreglos.cpp

#include<iostream.h>

float  promedio( float arreglo[], int tamaño ) {
  float suma = 0;
  for( int j = 0; j < tamaño; j++ )
    suma = suma + arreglo[j];
  return( suma/tamaño );
};

main() {
  float alumno1[3]={9.7, 8.2, 7.0};
  float alumno2[5]={5.9, 10.0, 8.5, 9.1, 6.2};
  cout << "Promedio del alumno 1 es:"
       << promedio( alumno1, 3 );
  cout << "\n Promedio del alumno 2 es:"
       << promedio( alumno2, 5 );

  return 0;
}
```

Ejercicio 5.7.- Hacer un programa que llame a la función factorial para desplegar en pantalla el factorial de un número proporcionado por el usuario. El programa debe pedir números y desplegar su factorial hasta que el usuario oprima "ESC". Proteger el programa para que no se introduzca un número mayor a 15.

En la siguiente solución parcial se muestra como el programa solicita datos al usuario y entrega el factorial correspondiente al dato de entrada indefinidamente, el proceso solo se detiene cuando el usuario oprime “ESC”. ¿Como terminaríamos de definir la función `factorial`?

Solución parcial:

```
long int factorial( <parametro> ) {
  < cuerpo de la función >
};

main() {
  long int resultado; int dato;
  char c, cr[2];
  do
  {
    cout << "De que número quieres el factorial?";
    cin >> dato;
    if( dato > 15 )
      cout << "El factorial de " << dato << " es muy grande"
           << " da un numero más pequeño! "
           << "(oprime cualquier tecla) \n";
    else {
      resultado = factorial( dato );
      cout << dato <<" factorial es: " << resultado
           << "\n oprime ""ESC"" para salir \n";
    };
    c = getch();
    cin.getline(cr,2);
  }while( 27 != c );

  return 0;
}
```

Solución completa:

```
// Obtiene el factorial de números menores o iguales a 15
//  funFactorial.cpp

#include<iostream.h>

long int factorial( int n ) {
  long int fact = 1;
  for( int i = n; i > 1; i-- )
    fact = fact * i;

  return fact;
};

main() {
  long int resultado; int dato;
  char c, cr[2];
  do
  {
    cout << "De que número quieres el factorial?";
    cin >> dato;
    if( dato > 15 )
      cout << "El factorial de " << dato << " es muy grande"
           << " da un numero más pequeño! "
           << "(oprime cualquier tecla) \n";
    else {
      resultado = factorial( dato );
      cout << dato <<" factorial es: " << resultado
           << "\n oprime ""ESC"" para salir \n";
    };
    c = getch();
    cin.getline(cr,2);
  }while( 27 != c );

  return 0;
}
```

V.6 Uso de funciones para hacer menús

Un ciclo infinito es útil en la programación. Por ejemplo, podemos hacer que se despliegue un menú para que el usuario trabaje con las opciones que necesite las veces que lo requiera, sin necesidad de correr el programa una, otra y otra vez. El ciclo infinito se hace con la instrucción: `while(1)`. Si la condición es un `1`, ésta siempre es verdadera entonces lo que esta dentro del `while` se ejecuta para siempre. La manera de salir del ciclo es con la instrucción `break`.

Ejemplo 5.5.- Hacer un programa con una función que convierta de grados Fahrenheit a grados centígrados y otra al revés:

C = (F – 32)*5 / 9

F = C*9 / 5 + 32

y que pregunte al usuario cuál conversión quiere hacer mediante un menú. El desplegado del menú se repite hasta que el usuario elija la opción “salir”.

En los ejercicios *5.3* y *5.4* de la sección V.4.5 ya tenemos resuelto por separado la conversión de centígrados a Farenheit y viceversa, ahora solo resta elaborar un menú dentro de un ciclo infinito de la siguiente forma:

```
// Programa para conversion de grados
// farenheit <--> centigrados
// TEMPERATURA.CPP
#include<iostream.h>

double F_a_Centigrados( double f ) {
  return( ( ( f - 32.0 ) * 5.0 ) / 9.0 );
};

double C_a_Farenheit(double c) {
  return( 9.0 * c / 5.0 + 32.0 );
};

int pideOpcion() {
  // elección y validación de opción
  int o;
  cout<<"  (1).- convertir de grados Farenheit a
                 centigrados.\n"
      <<"  (2).- convertir de grados centigrados a
                 Farenheit.\n"
      <<"  (3).- Para terminar el programa. \n \n";
  do {
    cout << "Elija una opcion del 1 al 3: ";
    cin  >> o;
  }while( ( o < 1 ) || ( o > 3 ) );

  return opcion;
}

void Caso1() {
  double dato, dato_convertido;

  cout<<"\nIntroduzca la temperatura en grados Farenheit: ";
  cin>>dato;
  dato_convertido = F_a_Centigrados( dato );

  cout<<endl<<dato<<" grados Farenheit equivalen a: "
      <<dato_convertido<<" grados centigrados \n";
}

void Caso2() {
  double dato, dato_convertido;

  cout<<"\nIntroduzca la temperatura en grados Centigrados:";
  cin>>dato;
  dato_convertido = C_a_Farenheit( dato );

  cout<<endl<<dato<<" grados centigrados equivalen a: "
      <<dato_convertido<<" grados farenheit \n";
}

main() {
  int opcion;
  cout<<" Programa para convertir unidades de temperatura. \n ";

  // repetir lo siguiente
  while( 1 ) {
```

```
    opcion = pideOpcion();

    switch( opcion ) {
      case 1: Caso1();
              break; // rompe el switch
      case 2: Caso2();
              break; // rompe el switch
    }; // fin del switch

    if( opcion == 3 )
      break; // rompe el while( 1 )

  }; // end while( 1 )

  cout<<"\n adios....";
}
```

En este programa principal se observa que el usuario elige una de tres opciones, con la opción 1 convierte de Farenheit a Centígrados, con la opción 2 de Centígrados a Farenheit y con la 3 se sale del programa. La opción que se captura es siempre válida (entre 1 y 3) ya que la función `pideOpcion()` se encarga de ello "liberando" así al programa principal de esa responsabilidad, lo que lo hace más fácil de leer.

Capítulo VI Listas ligadas, pilas y colas.

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

Comprender los conceptos y elaborar programas que contengan:

- Listas ligadas
- Pilas
- Colas (filas)

VI.1 Listas ligadas

VI.1.1 Definición de lista ligada

Una lista ligada es un conjunto de elementos *ligados* o encadenados mediante *ligas* o enlaces, como se muestra en la Figura VI-1. Las *listas ligadas* son "cajas" con información que se alojan en la memoria dinámica, cada una de estas "cajas" se crea conforme se va necesitando. Cada vez que se aloja una nueva "caja" se liga a la caja anterior, de tal forma que se puede recorrer la lista siguiendo las ligas que unen una caja con otra. Las *listas ligadas* permiten trabajar con muchos o con muy pocos elementos usando solamente la memoria necesaria.

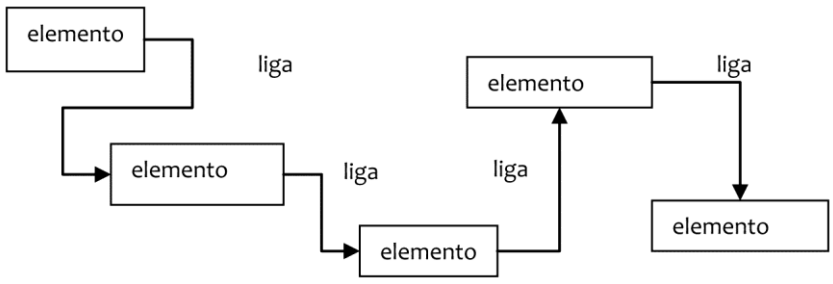


Figura VI-1 Ejemplo de lista ligada.

Es posible usar arreglos en lugar de listas ligadas, sin embargo cuando no se sabe cuántos elementos habrá se corre el riesgo de desperdiciar mucha memoria, o de que no quepan todos los elementos en el arreglo.

¿Cuándo son útiles las listas ligadas?

Cuando el tamaño de la lista es muy variable y se quiere optimizar el uso de la memoria. Si se tienen muy pocos elementos no tiene caso alojar mucha memoria, las listas ligadas permiten trabajar con muchos o con muy pocos elementos usando solamente la memoria necesaria.

La memoria que ocupan las **estructuras estáticas**: (*Arreglos, estructuras, arreglos de estructuras, estructuras con arreglos,* ….) se establece durante la compilación y no se altera durante la ejecución del programa.

La memoria que ocupan las **estructuras dinámicas** (*Listas ligadas que forman: pilas, colas, árboles, círculos, ...*) crece y se contrae conforme se ejecuta el programa.

VI.1.2 Construcción de una lista ligada

Para construir una *lista ligada* pensemos que vamos a utilizar una *caja* en donde se guardará un *elemento* y una *liga* o *enlace* a la siguiente *caja*, esta *caja* se representa en C/C++ mediante un registro (`struct`). Entonces necesitamos:

Definir el tipo de elemento que se guardará dentro de cada *caja* de la lista, por ejemplo: `int, float, double, struct`, etc.

Definir el enlace como un campo del registro (`struct`) el cual debe ser un apuntador a otra *caja.* Por ejemplo, para hacer una lista ligada de enteros habría que definir cada *caja* como un *registro* con la siguiente estructura:

```
struct s_caja {
  int    elemento;
  s_caja *enlace;
};
```

La lista se hace poniendo un elemento en una *caja nueva* y ligándola a alguna *caja*. Para agregar una *caja nueva* en la lista es necesario

- Alojar memoria de tipo *caja.*
- Poner los datos que llevará esta caja.
- Asignar un valor al *elemento* de la caja.
- Asignar un valor al *enlace* de la caja: el valor será la dirección de alguna caja o bien el valor de NULL.

El resultado de aplicar tres veces el procedimiento anterior podemos visualizarlo en la Figura VI-2.

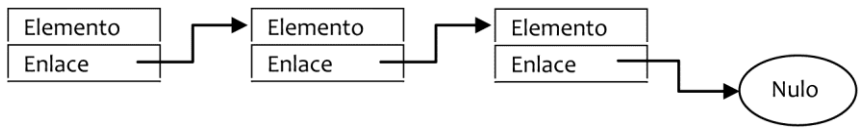


Figura VI-2 Lista ligada hecha con cajas representadas con registros.

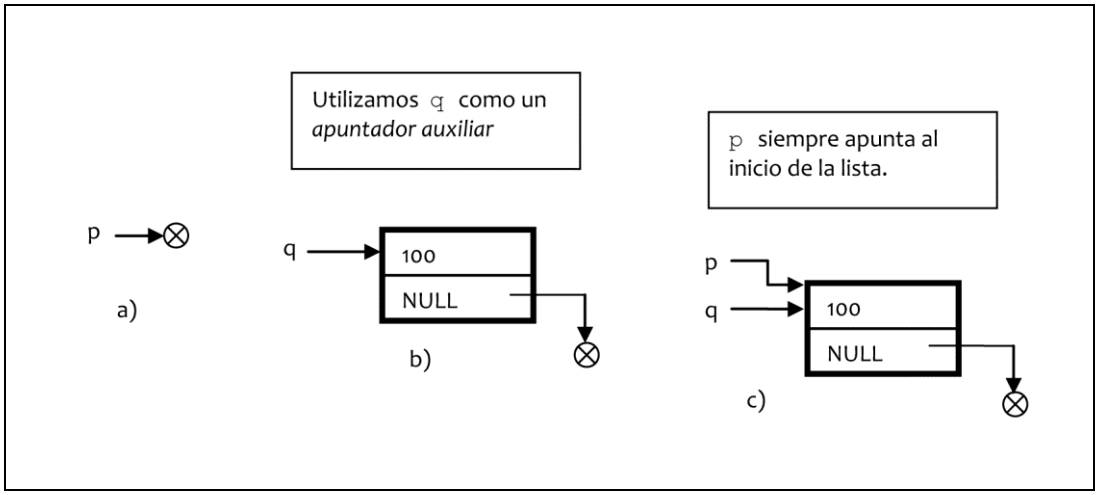
Al construir una lista ligada hay que tomar en cuenta los siguientes aspectos importantes:

- Hay que conservar siempre un apuntador hacia el inicio de la lista.
- Las cajas siempre deben estar apuntadas por algún apuntador.

Ejemplo 6.1.- Ilustraremos gráficamente como se va construyendo una lista ligada a partir del siguiente código de ejemplo:

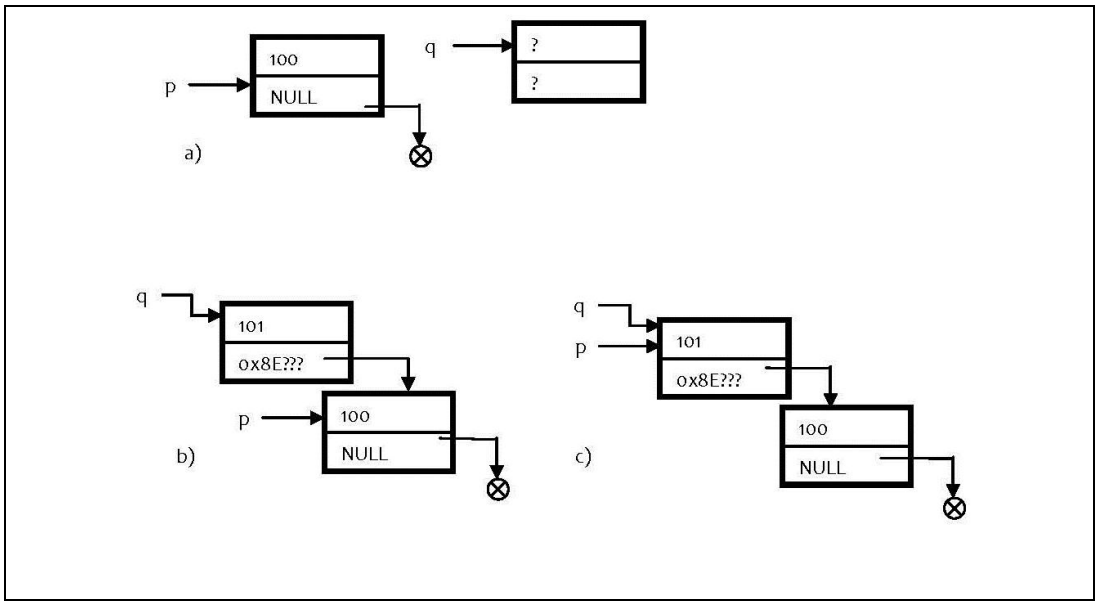
```
s_caja *p, *q;

p = NULL;

for( int i=0; i<3; i++ ) {
  q = new s_caja;
  q->elemento = 100+i;
  q->enlace = p;
  p = q;
};
```

Para `i=0`:



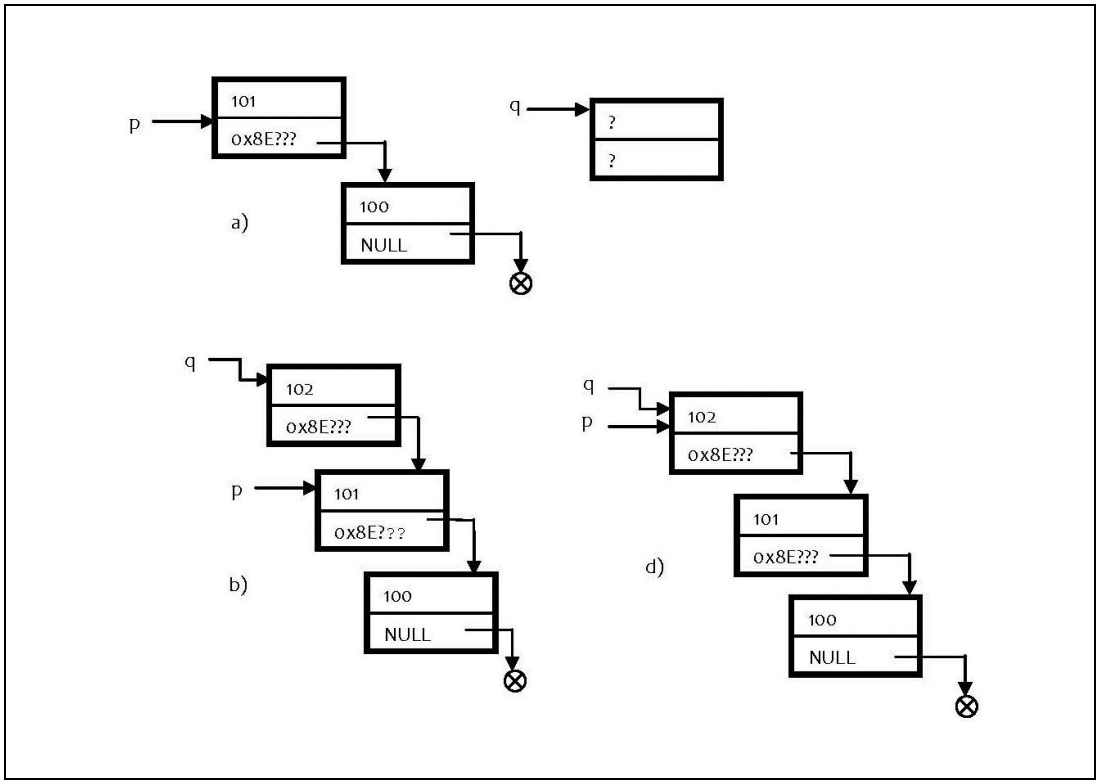

```
a) p = NULL

b) q = new s_caja
   q->elemento = 100+0
   q->enlace = p

c) p = q
```

Para `i=1`:




```
a) q = new s_caja

b) q->elemento = 100+1
   q->enlace = p

c) p = q
```

Para `i=2`:




```
a) q = new s_caja

b) q->elemento = 100+2
   q->enlace = p

c) p = q
```

VI.1.3 Desalojar la memoria ocupada por una lista ligada

Cuando ya no se va a usar una lista ligada, es necesario desalojar la memoria que ocupa, Al destruir una lista ligada hay que tomar en cuenta los siguientes aspectos importantes:

- Hay que conservar siempre un apuntador hacia el inicio de la lista.
- Usamos un apuntador auxiliar para no perder el apuntado al inicio de la lista.

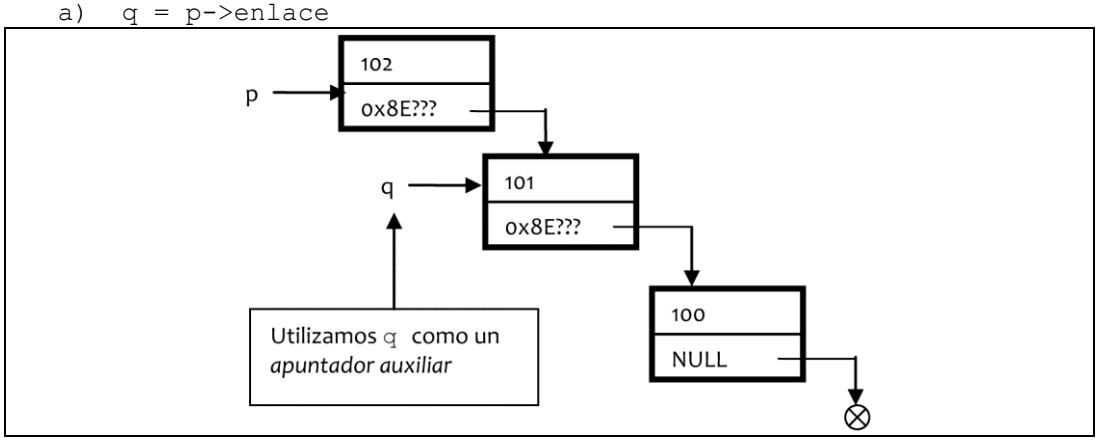
No es necesario saber cuantos elementos contiene la lista, las cajas de la lista se van desalojando hasta que el apuntador al inicio de la lista sea NULL.

Ejemplo 6.2.- Ilustraremos gráficamente como se va construyendo una lista ligada a partir del siguiente código de ejemplo:

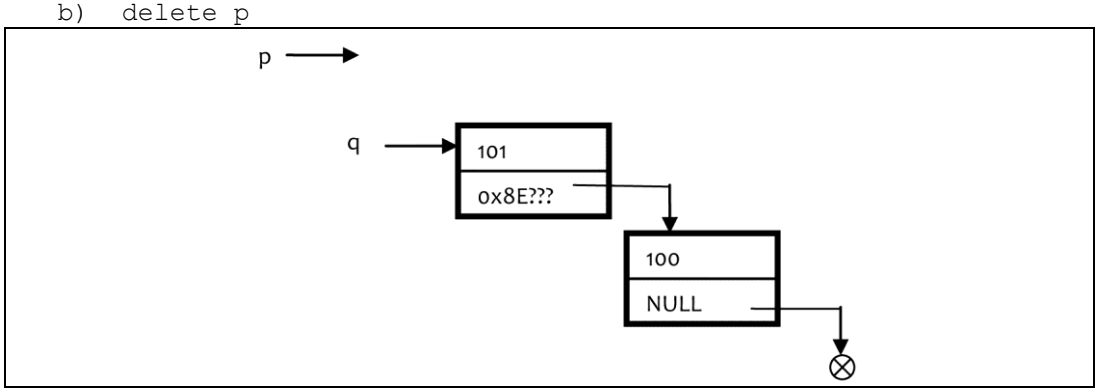
```
s_caja *p, *q;
// no es necesario saber
// cuántos elementos hay
while( p != NULL ) {
  q = p->enlace;
  delete p;
  p = q;
};
```

Primera iteración:

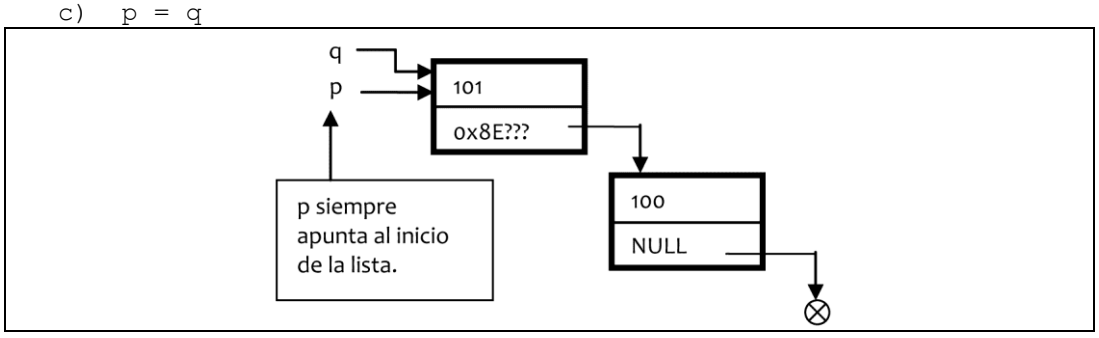
a) `q = p->enlace`



b) `delete p`

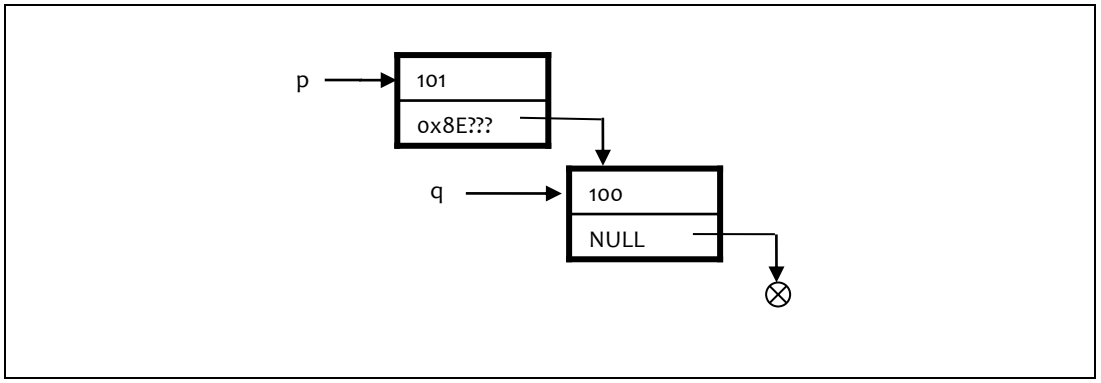


c) `p = q`

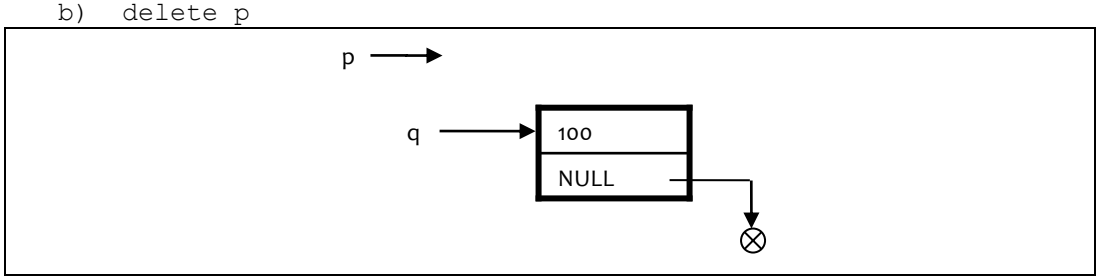


Segunda iteración:

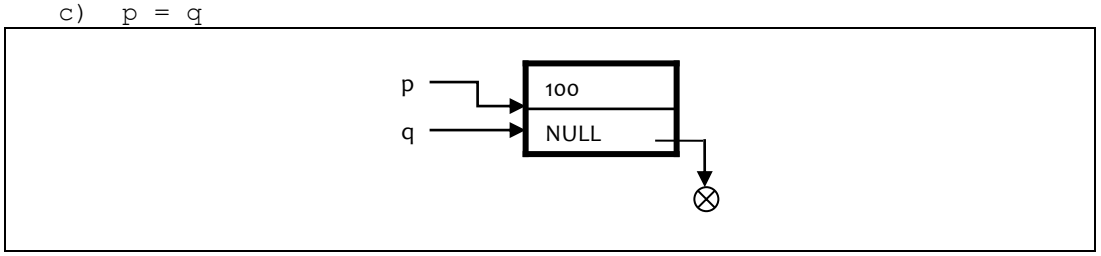
a) `q = p->enlace`



b) `delete p`

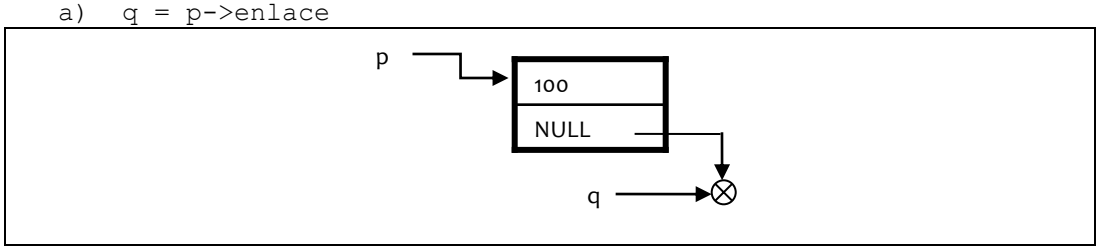


c) `p = q`

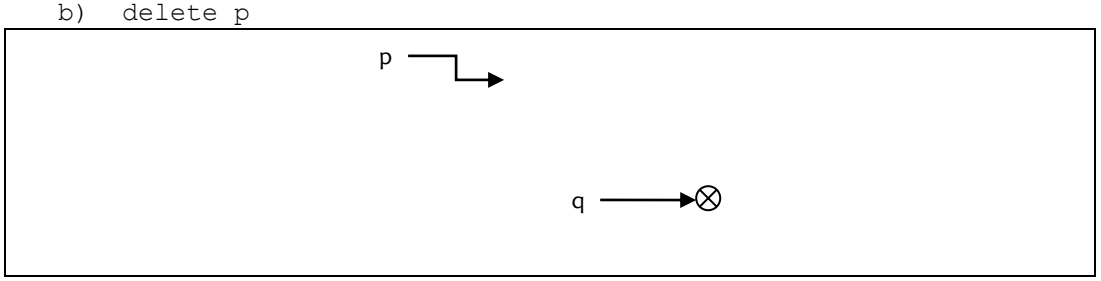


Tercera iteración:

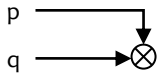
a) `q = p->enlace`



b) `delete p`



c) `p = q`



En este caso en particular en el que sólo había tres cajas en la lista ligada, ya no se lleva a cabo una cuarta iteración, porque en la tercera iteración ya se desalojaron todas las cajas y `p` apunta a `NULL`.

VI.1.4 Un *registro* o *estructura* como elemento de la lista ligada

El tipo del elemento que está dentro de la caja, puede ser una estructura, alternativamente, puede haber más de un elemento en cada caja. Por ejemplo, si el elemento es de tipo `s_alumno` y esta estructura contiene el nombre y el promedio de cada estudiante, entonces tendríamos lo siguiente:

```
struct s_alumno{
  char  nombre[30];
  float promedio;
};

struct s_caja {
  s_alumno elemento;
  s_caja  *enlace;
};
```

Ejemplo 6.3.- ¿Cuál sería el código para formar una lista ligada como la de la Figura VI-3?

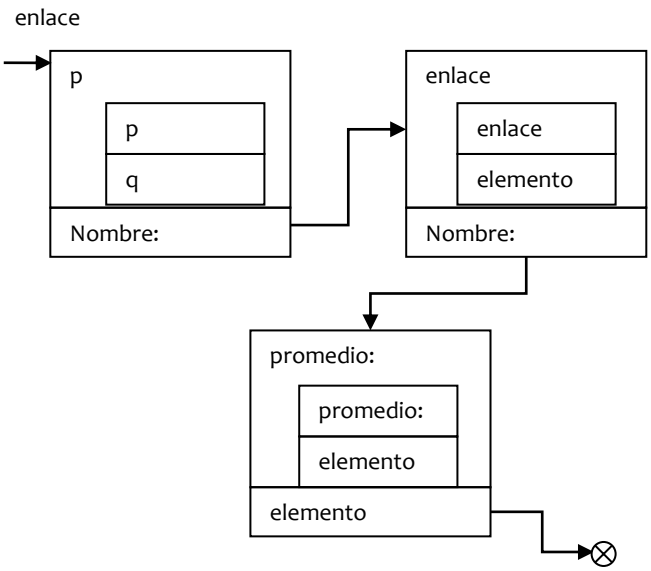


Figura VI-3 *Ejemplo de una lista ligada con registros (estrructuras).*

Solución: Aplicaremos el concepto de *diseño modular* del capítulo anterior, utilizando como funciones de apoyo las siguientes:

```
bool otra_caja()
{
  char c;
  cout<<"\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};
```

La función `otra_caja()` pregunta al usuario si tiene otra caja más, en caso afirmativo, regresa el valor `true`, y `false` en caso contrario.

La función `hacer_caja()` aloja memoria para una caja nueva, pide los datos al usuario y regresa un apuntador a la caja nueva.

```
s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;
  q = new s_caja;
  if( q == NULL )return NULL;

  // pide los datos
  cout<<"\n Nombre del alumno: ";
  cin.getline( caja_nueva->nombre, 20 );
  cout<<"\n Su promedio?";
  cin>> caja_nueva->promedio;
  cin.getline( cr, 2 );

  return q;
}
```

Para hacer la lista ligada de la *figura 6.3*, en donde el usuario pueda introducir un número variable de cajas, tenemos el siguiente programa principal:

```
main() {
  char  cr[2];
  s_caja *caja_nueva   = NULL,
         *caja_inicial = NULL,
         *caja_aux     = NULL;

  // Construye una lista ligada
  while( otra_caja( ) ) {
    caja_nueva = hacer_caja( );
    caja_nueva->sig_caja = caja_inicial;
    caja_inicial = caja_nueva;
  };

// Escribe los datos contenidos en cada una de las cajas
  // recorre la lista ligada (sin borrarla)
  caja_aux = caja_inicial;
  while( caja_aux != NULL) {
    cout << endl << endl
         << "\n El nombre del alumno es: "
         << caja_aux->nombre;
    cout << "\n Su promedio es : " << caja_aux->promedio;
    caja_aux = caja_aux->sig_caja;
  };

// Libera la memoria ocupada por los datos de todas las cajas
  // recorre y deshace la lista ligada
  caja_aux = caja_inicial;
  while( caja_aux != NULL) {
    caja_aux = caja_aux->sig_caja;
    delete caja_inicial;
    caja_inicial = caja_aux;
  };

  return 0;
}
```

En este programa se incluye además, el código para desplegar el contenido de la lista ligada y posteriormente el código para desalojar la memoria ocupada por la lista.

VI.2 Pilas y colas

VI.2.1 Concepto de pila

En una *pila* se almacenan elementos uno sobre otro, de tal forma que el primer elemento disponible es el último que fue almacenado.

A una pila también se le conoce como *lista* LIFO (Last Input First Output), es decir, el primero en entrar es el último en salir. Los ejemplos de lista ligadas de la sección anterior son precisamente de tipo LIFO, es decir, son pilas, ya que cada que se inserta un nuevo elemento se hace al principio de la lista, de tal forma que

el primer elemento disponible es el último que se introdujo. Una pila en la programación puede compararse con una pila de platos o de tortillas. Como puede observarse en la Figura VI-4 el último elemento que se coloca dentro de una pila es el primero en salir, ya que éste se encuentra hasta arriba de la pila.

Figura VI-4 *En una pila el último elemento que se coloca es el primero en salir.*

Existen dos operaciones básicas para trabajar con pilas, que son las funciones `push()` y `pop()`.

VI.2.1.1 Las operaciones PUSH y POP

Normalmente, cuando se trabaja con pilas, se utiliza una función llamada `push( )`, que en inglés significa *empujar*, y se encarga de poner hasta arriba de la pila a una nueva caja, recibe como parámetro un apuntador a la caja nueva de tal manera que el apuntador al inicio de la pila apunta hacia el nuevo elemento enviado. Si llamamos `inicio` al apuntador que apunta al principio de la pila y hacemos que `inicio` sea una *variable global*, entonces, el código de la función `push( )` es el siguiente:

```
void push( s_caja *q ) {
  if( q == NULL )
    return;
  // "empuja" una caja
  q->siguiente = inicio;
  inicio = q;
}
```

La otra operación básica cuando se trabaja con pilas es la función `pop( )`, que en este contexto puede traducirse como *expulsar* o *sacar* de la pila. La función `pop( )` entrega un apuntador al elemento *expulsado* de la pila y hace que la variable global `inicio` de tipo apuntador apunte a la siguiente caja de la pila. El código de la función `pop( )` es el siguiente:

```
s_caja *pop() {
  // saca una caja
  s_caja *q;
  q = inicio;
  if (inicio != NULL)
    inicio = inicio->siguiente;
  return q;
}
```

Ejemplo 6.4.- En la Figura VI-5 se ilustra como trabaja la función `push()` en una pila. Cuando se crea una caja nueva apuntada por el apuntador `q` entonces al hacer **`push`**`(q)` la caja nueva se *empuja* y queda al inicio de la pila.

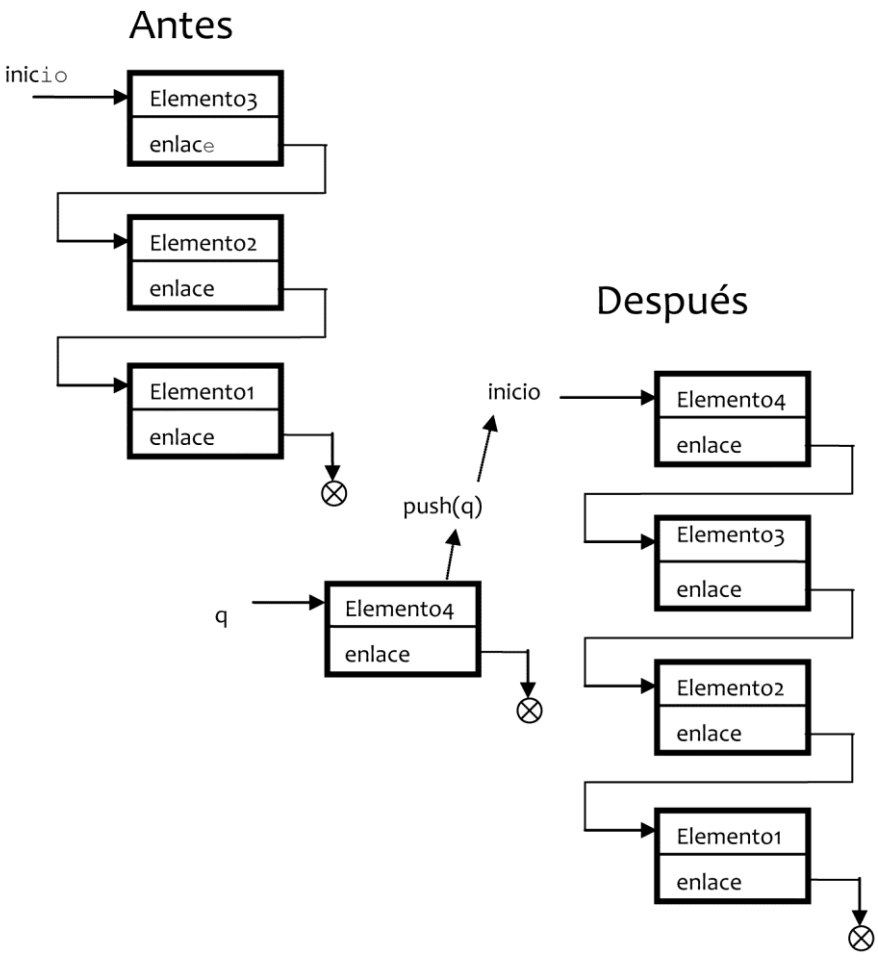


Figura VI-5 *Cómo trabaja la función `push()` en una pila.*

Ejemplo 6.5.- En la Figura VI-6 se ilustra como trabaja la función `pop()` en una pila. Al hacer `q = pop();` el apuntador `q` apunta a la caja que estaba hasta arriba de la pila, y a partir de este momento `inicio` apunta a la siguiente caja de la pila.

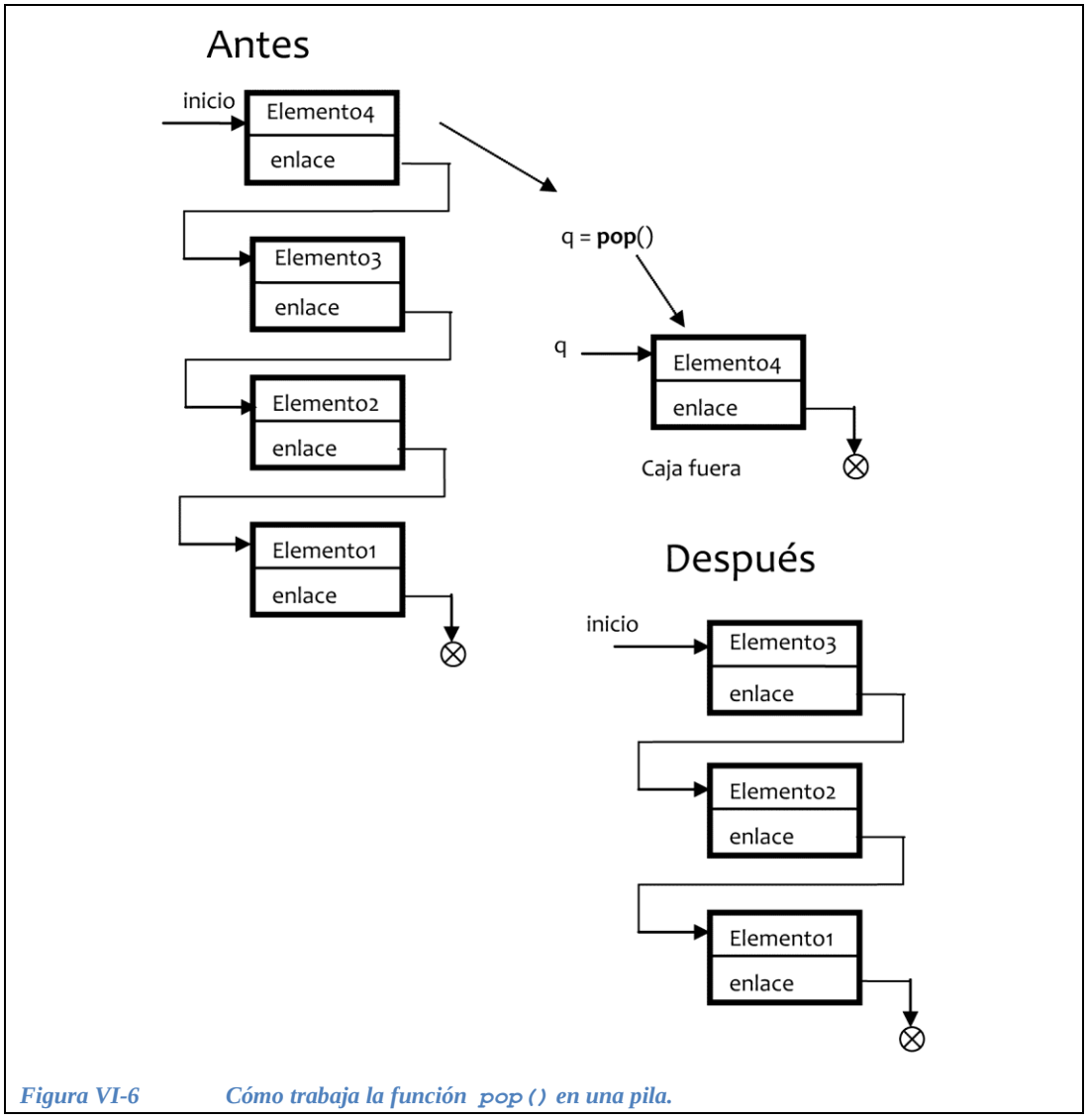


Figura VI-6 *Cómo trabaja la función `pop()` en una pila.*

Ejemplo 6.6.- A continuación presentamos un ejemplo de un programa que utiliza las funciones descritas en este capítulo para:

- Crear y guardar cajas en una pila.
- Sacar, usar y desechar las cajas de la pila.

```
main() {
  // ...
  // crear y guardar
  while( otra_caja() )
    push( hacer_caja() );

  // sacar, usar y desechar
  while( NULL != (q = pop())) {
    // trabajar con la caja
    ::
    delete q;
  };

  // …
}
```

Crear y Guardar cajas en la pila

Sacar cajas de la pila y usarlas

Si ya no se usan se borran

VI.2.2 Concepto de fila o cola

En una *fila* o *cola* el primer elemento que se saca de la lista es el primero que entró. A una *cola* también se le conoce como *lista* FIFO (First Input First Output), es decir, el primero en entrar es el primero en salir. Una fila o cola en la programación funciona como una fila del banco o del supermercado. Como puede observarse en la Figura VI-7 el primer elemento que entra en la fila es el primero en salir.

Figura VI-7 El primer elemento que entra a la fila es el primero en salir

Es importante mencionar que para trabajar con filas es necesario tener dos apuntadores, uno al `inicio` de la fila y otro al `fin` de ésta.

Existen dos operaciones básicas para trabajar con filas (también llamadas colas), que son las funciones `formar( )` y `despachar( )`.

VI.2.2.1 Las operaciones FORMAR y DESPACHAR

Normalmente, cuando se trabaja con filas, se utiliza una función llamada `formar()` que se encarga de poner al final de la cola una nueva caja, recibe como parámetro un apuntador a la caja nueva de tal manera que el apuntador que apunta al final de la cola ahora apuntará hacia el nuevo elemento enviado. Si llamamos `fin` al apuntador que apunta al final de la fila y hacemos que `fin` sea una *variable global*, entonces, el código de la función `formar()` es el siguiente:

```
void formar( s_caja *q ) {
  if( q == NULL )
    return;

  q->siguiente = NULL;

  // "forma" una caja al final
  if( fin == NULL )
    caja_inicial = q;

  else
    fin->siguiente = q;

  fin = q;
}
```

La otra operación básica cuando se trabaja con filas es la función `despachar()` que saca de la fila al primer elemento que esta formado. La función `despachar()` entrega un apuntador al elemento *despachado* de la cola y hace que la variable global `inicio` de tipo apuntador apunte a la siguiente caja de la fila (cola). El código de la función `despachar()` es el siguiente:

```
s_caja *despachar() {
  // si ya está vacía
  if( inicio == NULL )
    return NULL;

  s_caja *q;
  q = inicio;
  inicio = inicio->siguiente;
  q->siguiente = NULL;

  // si queda vacía
  if( inicio == NULL )
    fin = NULL;

  return q;
}
```

Ejemplo 6.7.- En la Figura VI-8 se ilustra como trabaja la función `formar()` en una cola. Cuando se crea una caja nueva apuntada por el apuntador `q` entonces al hacer **`formar`**`(q)` la caja nueva se *forma* y queda al final de la cola.

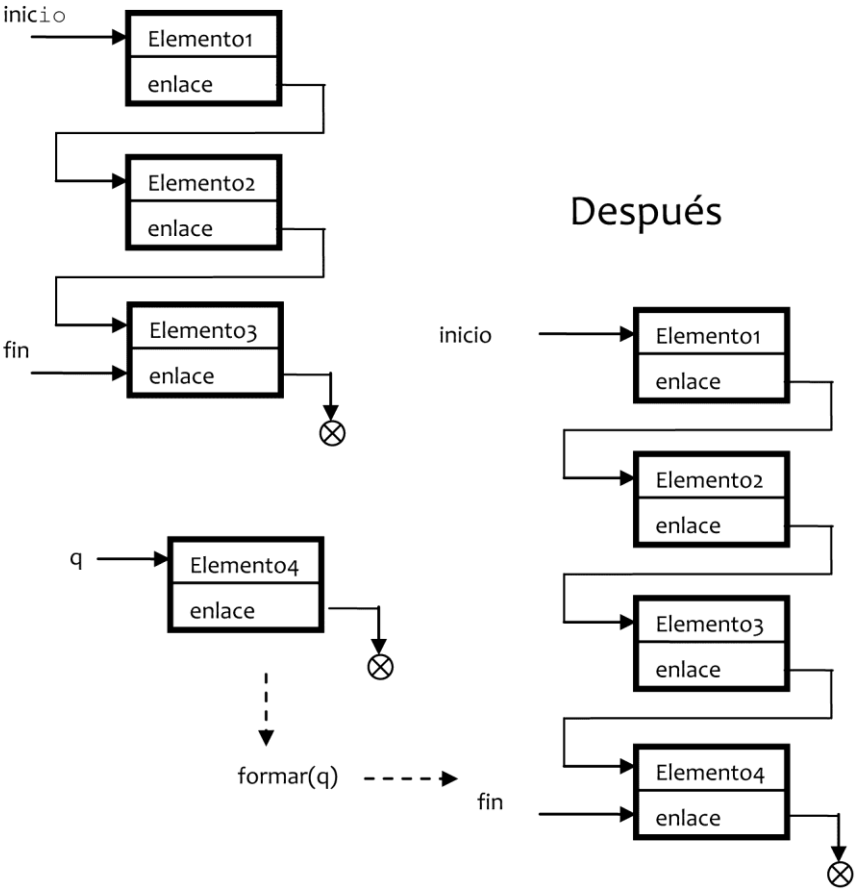


Figura VI-8 *Cómo trabaja la función `formar()` en una cola.*

Ejemplo 6.8.- En la Figura VI-9 se ilustra como trabaja la función `despachar()` en una cola. Al hacer `q = despachar()` el apuntador `q` apunta a la caja que esta al principio de la cola, y a partir de este momento `inicio` apunta a la caja que estaba formada en segundo lugar en la cola.

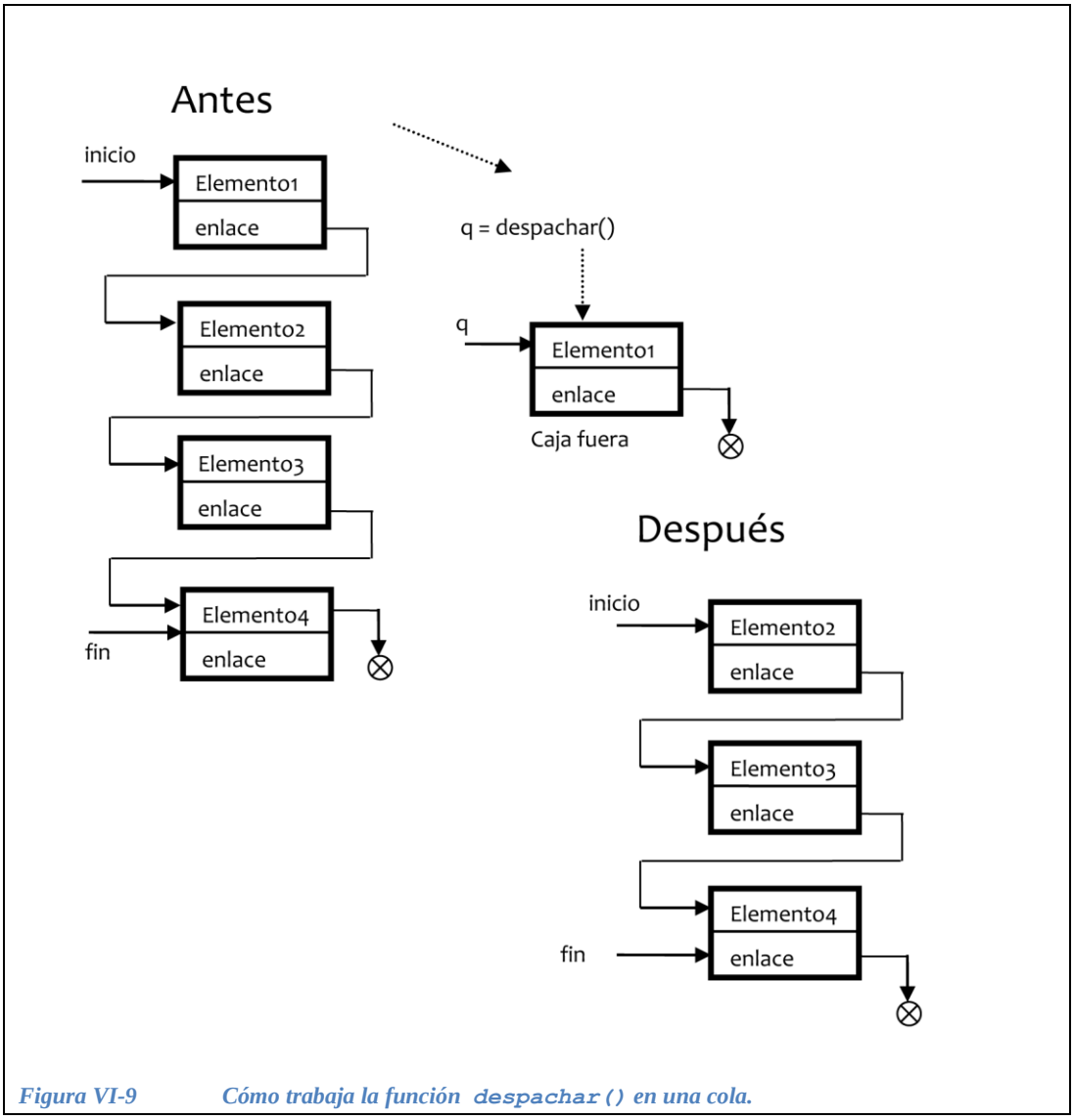
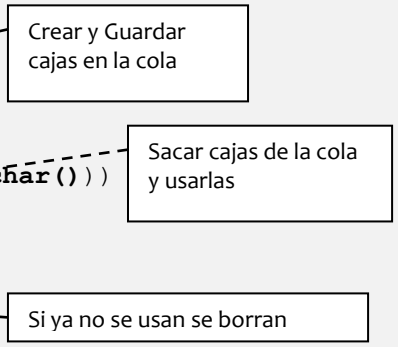


Figura VI-9 *Cómo trabaja la función* `despachar()` *en una cola.*

Ejemplo 6.9.- A continuación presentamos un ejemplo de un programa que utiliza las funciones descritas en este capítulo para:

1. Crear y guardar cajas en una fila o cola.
2. Sacar, usar y desechar las cajas de la cola.

```
3. main() {
4.   // …
5.   // crear y guardar
6.   while( otra_caja() )
7.     formar( hacer_caja() );
8.
9.   // sacar, usar y desechar
10.     while( NULL != (q = despachar()))
11.       // trabajar con la caja
12.       ::
13.       delete q;
14.     };
15.
16.     // …
17.   }
```



VI.2.3 Ejercicios de pilas y colas

Ejercicio 6.1.- Hacer un programa que genere una pila de 3 elementos donde cada elemento guarda un double (además del enlace). Inicializar cada elemento con cero. Escribir en pantalla todos los elementos de la pila.

Solución parcial:

```
#include <iostream>

int main() {
  struct s_caja {
    double    elemento;
    s_caja *enlace;
  };
  s_caja *p, *q;

  p = NULL;
  for( int i=0; i<3; i++ ) {
    q = new ...;
    q->elemento = . . .;
    q->enlace = . . .;
    p = . . .;
  };

  // Recorrido de la lista
  int j=1;
  q = . . . ;
  while( . . . ){
    cout << "El elemento " << j << " es: "
         << q->elemento << endl;
    q = . . . ;
    j++;
  };

  // Desalojar la lista
  while( p != NULL ) {
    q = . . .;
    delete p;
    . . .;
  };

  return 0;
}
```

Solución completa:

```
#include <iostream>

int main() {
  struct s_caja {
    double    elemento;
    s_caja *enlace;
  };
  s_caja *p, *q;

  p = NULL;
  for( int i=0; i<3; i++ ) {
    q = new s_caja;
    q->elemento = 0;
    q->enlace = p;
    p = q;
  };

  // Recorrido de la lista
  int j=1;
  q = p;
  while( q!= NULL ){
    cout << "El elemento " << j << " es: "
         << q->elemento << endl;
    q = q->enlace;
    j++;
  };

  // Desalojar la lista
  while( p != NULL ) {
    q = p->enlace;
    delete p;
    p = q;
  };

  return 0;
}
```

Ejercicio 6.2.-. Hacer un programa que pida al usuario una serie de números enteros y que conforme los reciba, los guarde en una pila. Recorrer la lista y desplegar sus elementos en pantalla. Desalojar la pila. El programa termina cuando el usuario proporciona un número negativo.

Primera solución parcial:

```
main() {
  struct s_caja {
    int    elemento;
    s_caja *enlace;
  };
  s_caja *p = NULL, *q;
  int elem;

  cout<<"Para terminar, introduce un número negativo \n";
  do {
    cout << "dame un número entero: ";    cin>>elem;
  }while (elem >= 0);

  // Recorrido de la lista
  ...

  // Desalojar la lista
  ...

  return 0;
}
```

Segunda solución parcial:

```
main() {
  struct s_caja {
    int    elemento;
    s_caja *enlace;
  };
  s_caja *p = NULL, *q;
  int elem;

  cout << "Para terminar, introduce un número negativo \n";
  do {
    cout << "dame un número entero: ";    cin>>elem;
    if( elem >= 0 ) {
      q = new s_caja;
      q->elemento = elem;
      q->enlace = p;
      p = q;
    };
  }while( elem >= 0 );

  // Recorrido de la lista
  int j=1;
  q = p;

  // Desalojar la lista
  ...

  return 0;
}
```

Solución completa:

```
main() {
  struct s_caja {
    int    elemento;
    s_caja *enlace;
  };
  s_caja *p = NULL, *q;
  int elem;

  cout << "Para terminar, introduce un número negativo \n";
  do{
    cout << "dame un número entero: ";    cin>>elem;
    if( elem >= 0 ) {
      q = new s_caja;
      q->elemento = elem;
      q->enlace = p;
      p = q;
    };
  }while( elem >= 0 );

  // Recorrido de la lista
  int j=1;
  q = p;
  while( q!= NULL ) {
    cout << "El elemento " << j << " es: "
         << q->elemento << endl;
    q = q->enlace;
    j++;
  };

  // Desalojar la lista
  while( p != NULL ) {
    q = p->enlace;
    delete p;
    p = q;
  };

  return 0;
}
```

Ejercicio 6.3.- Modificar el programa anterior para que en lugar de que pida al usuario un número entero, pida un nombre y una calificación guardando la información en la pila. Utilizar la siguiente estructura:

```
struct s_alumno{
  char  nombre[30];
  float promedio;
};
```

La instrucción para el usuario es:

```
Oprime "S" si  deseas agregar otro elemento.
```

En el caso de que el usuario oprima cualquier otra tecla, el programa despliega todos los elementos de la pila, la desaloja y termina.

Recordar que el tipo de elemento que se guarda en una caja, puede ser una estructura.

```
struct s_caja {
  tipo    elemento;
  s_caja *enlace;
};
```

Recordar que para capturar la cadena de caracteres en C++ se utiliza:

```
cin.getline( <cadenaDestino>, <longitud_Cadena> );
```

En este caso:

```
cin.getline( q->elemento.nombre, 30 );
```

Después de capturar un número hay que recoger la basura del buffer con otro `cin.getline`, en este caso:

```
cin >> q->elemento.promedio;
cin.getline( cr, 2 );
```

Solución parcial:

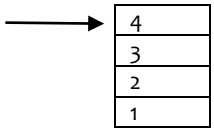
```
#include <iostream>
#include <conio.h>

int main() {
  struct s_alumno{
    char  nombre[30];
    float promedio;
  };
  struct s_nodo {
    s_alumno    elemento;
    s_nodo *link;
  };

  s_nodo *inicio, *q;
  int elem;
  char c, cr[2];

  inicio = NULL;
  do {
    cout << "Oprime ""S"" para agregar otro elemento \n";
    c = getch();
    if( c == 's' || c == 'S' ) {
      q = . . .;
      cout << "\n Nombre: ? ";
      cin.getline(. . ., 30);
      cout << "\n Promedio: ? ";
      cin >> . . . ;
      cin.getline(cr,2);
      q->link = . . .;
      inicio = . . .;
    };
  }while( c == 's' || c == 'S' );

  // Recorrido de la lista
  q = . . . ;
  while(. . . ) {
    cout << . . .  << " tiene de promedio: "
         << . . .  << endl;
    q = . . .;
  };

  // Desalojar la lista
  while( inicio != NULL ) {
    q = inicio->link;
    delete . . .;
    inicio = . . .;
  };

  return 0;
}
```

Solución completa:

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

int main() {
  struct s_alumno{
    char  nombre[30];
    float promedio;
  };
  struct s_nodo {
    s_alumno    elemento;
    s_nodo *link;
  };

  s_nodo *inicio, *q;
  int elem;
  char c, cr[2];

  inicio = NULL;
  do{
    cout << "Oprime ""S"" para agregar otro elemento \n";
    c = getch();
    if( c == 's' || c == 'S' ) {
      q = new s_nodo;
      cout << "\n Nombre: ? ";
      cin.getline(q->elemento.nombre, 30);
      cout << "\n Promedio: ? ";
      cin >> q->elemento.promedio;
      cin.getline(cr,2);
      q->link = inicio;
      inicio = q;
    };
  }while( c == 's' || c == 'S' );

  // Recorrido de la lista
  q = inicio;
  while( q!= NULL ) {
    cout << q->elemento.nombre << " tiene de promedio: "
         << q->elemento.promedio << endl;
    q = q->link;
  };

  // Desalojar la lista
  while( inicio != NULL ) {
    q = inicio->link;
    delete inicio;
    inicio= q;
  };

  return 0;
}
```

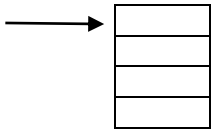
Ejercicio 6.4.- Hacer un programa que haga una pila llamada PilaA de enteros. Una vez que PilaA esta construida, sacar sus elementos y guardarlos en otra pila llamada PilaB. Si los elementos contenidos en PilaA son:

PilaA



¿Cómo quedan los elementos en Pila B?

PilaB



Sugerencia: utilizar dos apuntadores axiliares `*q1` y `*q2` para poder recorrer las dos pilas al mismo tiempo.

Solución parcial:

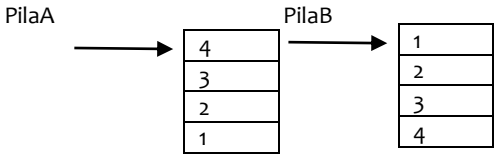
```
#include <iostream>
#include <conio.h>

int main() {
  struct s_caja {
    int    elemento;
    s_caja *enlace;
  };

  s_caja *pilaA, *pilaB, *q1, *q2;
  int elem;
  char c;

  pilaA = pilaB = NULL;
  do {
    cout << "Oprime ""S"" si deseas agregar otro elemento \n";
    c = getch();
    if( c == 's' || c == 'S' ) {
      cout << "dame un número entero: ";
      cin>>. . .;
      q1 = . . .;
      q1->elemento = elem;
      q1->enlace = . . .;
      . . . = q1;
    };
  }while ( c == 's' || c == 'S' );

  // Copiar los elementos de la pilaA a la pilaB
  q1 = pilaA;
  q2 = pilaB;
  while (q1!= NULL){
    q2 = . . .;
    q2->elemento = . . .;
    q2->enlace = . . .;
    pilaB =  . . .;
    . . . = q1->enlace;
  };

  // Desplegar los elementos de ambas pilas
  int j=1;
  q1 = pilaA;
  while( . . . ){
    cout << "El elemento " << j << " de la pila A es: "
         << . . . << endl;
    q1 = . . .;
    j++;
  };

  j=1;
  q2 = pilaB;
  while( . . . ){
    cout << "El elemento " << j << " de la pila B es: "
         << . . . << endl;
    . . .= q2->enlace;
    j++;
  };
```

```
  // Desalojar las listas
  while( . . . ) {
    q1 = pilaA->enlace;
    delete . . .;
    pilaA = . . .;
  };

  while( pilaB != NULL ) {
    . . .= pilaB->enlace;
    delete pilaB;
    pilaB = q1;
  };

  return 0;
}
```

Solución completa:

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

int main() {
  struct s_caja {
    int    elemento;
    s_caja *enlace;
  };

  s_caja *pilaA, *pilaB, *q1, *q2;
  int elem;
  char c;

  pilaA = pilaB = NULL;
  do {
    cout << "Oprime ""S"" si deseas agregar otro elemento \n";
    c = getch();
    if( c == 's' || c == 'S' ) {
      cout << "dame un número entero: ";    cin>>elem;
      q1 = new s_caja;
      q1->elemento = elem;
      q1->enlace = pilaA;
      pilaA = q1;
    };
  }while( c == 's' || c == 'S' );

  // Copiar los elementos de la pilaA a la pilaB
  q1 = pilaA;
  q2 = pilaB;
  while( q1!= NULL ){
    q2 = new s_caja;
    q2->elemento = q1->elemento;
    q2->enlace = pilaB;
    pilaB = q2;
    q1 = q1->enlace;
  };

  // Desplegar los elementos de ambas pilas
  int j=1;
  q1 = pilaA;
  while( q1!= NULL ) {
    cout << "El elemento " << j << " de la pila A es: "
         << q1->elemento << endl;
    q1 = q1->enlace;
    j++;
  };

  j=1;
  q2 = pilaB;
  while( q2!= NULL ) {
    cout << "El elemento " << j << " de la pila B es: "
         << q2->elemento << endl;
    q2 = q2->enlace;
    j++;
  };

  // Desalojar las listas
```

```
  while( pilaA != NULL ) {
    q1 = pilaA->enlace;
    delete pilaA;
    pilaA = q1;
  };

  while( pilaB != NULL ) {
    q1 = pilaB->enlace;
    delete pilaB;
    pilaB = q1;
  };

  return 0;
}
```



Ejercicio 6.5.- Modificar el siguiente programa para que haga una pila de enteros y que invierta su contenido. Desplegar el contenido inicial y después en el orden invertido. Agregar el código que determina la longitud de la pila.

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

struct s_caja {
  . . . dato;
  s_caja *siguiente;
};

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;
  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide los datos
  cout << " proporciona un entero: ";
  cin >> . . .;

  return . . .;
}

s_caja *Pop(s_caja* &inicio) {
  // saca una caja
  s_caja *q;
  q = inicio;
  if( inicio != NULL)
    inicio = . . .;
  return . . .;
}

void Push( s_caja *q, s_caja * . . . ) {
  if( q == NULL )
    return;
  // "empuja" una caja
  q->siguiente = . . .;
  inicio = . . .;
}

int main() {
  s_caja *q = NULL, *p = NULL;
  s_caja *iniPila1 = NULL, *iniPila2 = NULL;

  // crear y guardar elementos de la cola 1
  cout << "Proporciona los elementos de la pila \n";
  while( otra_caja() ){
    q = . . .;
    Push( q, iniPila1 );
  };
```

```
  cout << " \n La longitud de la pila es: " << … << endl;

  cout<<"La pila inicialmente tiene los siguientes elementos: \n";

  q = Pop(. . .);

  while( . . . ) {
    cout << q->dato << endl;
    Push( q, iniPila2);
    q = . . .;
  };

  cout << " Los elementos invertidos son: \n";
  q = Pop(iniPila2);
  while( . . . ) {
    cout << q->dato << endl,
    q = . . . ;
  };

  return 0;
}
```

Solución completa:

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

struct s_caja {
  int dato;
  s_caja *siguiente;
};

int j= 0;

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;
  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide los datos
  cout << " proporciona un entero: ";
  cin >> q->dato;

  return q;
}

s_caja *Pop(s_caja* &inicio) {
  // saca una caja
  s_caja *q;
  q = inicio;
  if( inicio != NULL)
    inicio = inicio->siguiente;
  return q;
}

void Push( s_caja *q, s_caja * &inicio ) {
  if( q == NULL )
    return;
  // "empuja" una caja
  q->siguiente = inicio;
  inicio = q;
  j++;
}

int main() {
  s_caja *q = NULL, *p = NULL;
  s_caja *iniPila1 = NULL, *iniPila2 = NULL;

  // crear y guardar elementos de la cola 1
  cout << "Proporciona los elementos de la pila \n";
  while( otra_caja() ) {
    q = hacer_caja();
```

```
    Push( q, iniPila1 );
  };

  cout << " \n La longitud de la pila es: " << j << endl;

  cout<< "la pila inicialmente tiene los siguientes elementos:\n";
  q = Pop(iniPila1);
  while( q != NULL ) {
    cout << q->dato << endl;
    Push( q, iniPila2);
    q = Pop(iniPila1);
  };

  cout << " Los elementos invertidos son: \n";
  q = Pop(iniPila2);
  while( q != NULL ) {
    cout << q->dato << endl,
    q = Pop(iniPila2);
  };

  return 0;
}
```

Ejercicio 6.6.- Completar el programa que construye una cola. Hacer que se calcule y despliegue su longitud antes de despachar sus elementos.

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

struct s_caja{
  char   nombre[20];
  float  precio;
  s_caja *siguiente;
};

s_caja *caja_inicial = NULL, *caja_final = NULL;
int j= 0;

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;
  char cr[2];
  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide los datos
  cout << " Nombre del producto? ";
  cin.getline (. . ., 30);
  cout << " precio? ";
  cin >> . . .;
  cin.getline( cr, 2);

  return . . .;
}

void Formar( s_caja *q ) {
  if( q == NULL )
    return;
  q->siguiente = NULL;
  // "forma" una caja al final
  if( caja_final == NULL )
    caja inicial = q;
  else
    caja_final->siguiente = . . .;
  caja_final = . . .;
}

s_caja *Despachar() {
  // si ya está vacía
  if( caja_inicial == NULL )
    return NULL;
  s_caja *q;
  q = caja_inicial;
  caja_inicial = . . . ;
  q->siguiente = NULL;
```

```
  // si queda vacía
  if( caja_inicial == NULL )
    caja_final = . . .;

  return . . .;
}


int main() {
  s_caja *q;
  // crear y guardar
  while( . . . )
    Formar( hacer_caja() );

  cout << " \n La longitud de la cola es: " << j << endl;

  q = caja_inicial;
  // sacar, usar y desechar
  while( NULL != (q = . . .) ) {
    // trabajar con la caja
    cout << "\n el producto: " << . . .
         << " cuesta: " << . . . << endl;
    delete q;
  };

  return 0;
}
```

Solución completa:

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

struct s_caja {
  char   nombre[20];
  float  precio;
  s_caja *siguiente;
};

s_caja *caja_inicial = NULL, *caja_final = NULL;
int j= 0;

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;
  char cr[2];
  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide los datos
  cout << " Nombre del producto? ";
  cin.getline (q->nombre, 30);
  cout << " precio? ";
  cin >> q->precio;
  cin.getline( cr, 2);

  return q;
}

void Formar( s_caja *q ) {
  if( q == NULL )
    return;
  q->siguiente = NULL;
  // "forma" una caja al final
  if( caja_final == NULL )
    caja inicial = q;
  else
    caja_final->siguiente = q;
  caja_final = q;
  j++;
}

s_caja *Despachar() {
  // si ya está vacía
  if( caja_inicial == NULL )
    return NULL;
  s_caja *q;
  q = caja_inicial;
  caja_inicial = caja_inicial->siguiente;
```

```
  q->siguiente = NULL;

  // si queda vacía
  if( caja_inicial == NULL )
    caja_final = NULL;

  return q;
}

int main() {
  s_caja *q;
  // crear y guardar
  while( otra_caja() )
    Formar( hacer_caja() );

  cout << " \n La longitud de la cola es: " << j << endl;

  q = caja_inicial;
  // sacar, usar y desechar
  while( NULL != (q = Despachar()) ) {
    // trabajar con la caja
    cout << "\n el producto: " << q->nombre
         << " cuesta: " << q->precio << endl;
    delete q;
  };

  return 0;
}
```

Ejercicio 6.7.- Completa el siguiente programa para que construya una cola1 y una cola2, que concatene las dos colas y que despliegue los elementos concatenados.

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

struct s_caja {
  int     dato;
  s_caja *siguiente;
};

int j=0;

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;

  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide el dato
  cout << " Numero entero ? ";
  cin >> . . .;

  return q;
}

void Formar( s_caja *q,
             s_caja* &caja_inicial,
             s_caja* &caja_final ) {
  if( q == NULL )
    return;
  q->siguiente = NULL;
  // "forma" una caja al final
  if( caja_final == NULL )
    caja_inicial = q;
  else
    caja_final->siguiente = . . .;
  caja final = . . .;

}

s_caja *Despachar( s_caja* &caja_inicial, s_caja* &caja_final ) {
  // si ya está vacía
  if( caja_inicial == NULL )
    return NULL;
  s_caja *q;
  q = . . .;
  caja_inicial = caja_inicial->siguiente;
  q->siguiente = NULL;

  // si queda vacía
```

```
  if( caja_inicial == NULL )
    caja_final = NULL;

  return q;
}

int main() {
  s_caja *q;
  s_caja *inicioC1 = NULL, *inicioC2 = . . .,
         *finC1=NULL, *finC2=. . .;

  // crear y guardar elementos de la cola 1
  cout << "Proporciona los elementos de la cola 1 \n";
  while( otra_caja() ) {
    q = hacer_caja( );
    Formar( q, inicioC1, finC1 );
  };

  cout << " \n La longitud de la cola 1 es: " << j << endl;
  j=0;

  // crear y guardar elementos de la cola 2
  cout << "Proporciona los elementos de la cola 2 \n";
  while( . . . ){
    q = . . .;
    . . .};

  cout << " \n La longitud de la cola 2 es: " << j << endl;


  // Código para concatenar las colas
  . . . .

  // Desplegar los elementos concatenados
  q = inicioC1;
  cout << "\n Los elementos concatenados son: \n";
  while ( . . .){
    cout << ... << endl;
    q = . . .;
  };

  return 0;
}
```

Solución completa:

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

struct s_caja {
  int     dato;
  s_caja *siguiente;
};

int j=0;

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;

  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide el dato
  cout << " Numero entero ? ";
  cin >> q->dato;

  return q;
}

void Formar( s_caja *q,
             s_caja* &caja_inicial,
             s_caja* &caja_final ) {
  if( q == NULL )
    return;
  q->siguiente = NULL;
  // "forma" una caja al final
  if( caja_final == NULL )
    caja_inicial = q;
  else
    caja_final->siguiente = q;
  caja final = q;
  j++;
}

s_caja *Despachar( s_caja* &caja_inicial, s_caja* &caja_final ) {
  // si ya está vacía
  if( caja_inicial == NULL )
    return NULL;
  s_caja *q;
  q = caja_inicial;
  caja_inicial = caja_inicial->siguiente;
  q->siguiente = NULL;

  // si queda vacía
```

```
  if( caja_inicial == NULL )
    caja_final = NULL;

  return q;
}

int main() {
  s_caja *q;
  s_caja *inicioC1 = NULL, *inicioC2 = NULL,
         *finC1=NULL, *finC2=NULL;

  // crear y guardar elementos de la cola 1
  cout << "Proporciona los elementos de la cola 1 \n";
  while( otra_caja() ) {
    q = hacer_caja( );
    Formar( q, inicioC1, finC1 );
  };

  cout << " \n La longitud de la cola 1 es: " << j << endl;
  j=0;

  // crear y guardar elementos de la cola 2
  cout << "Proporciona los elementos de la cola 2 \n";
  while( otra_caja() ) {
    q = hacer_caja( );
    Formar( q, inicioC2, finC2 );};

  cout << " \n La longitud de la cola 2 es: " << j << endl;

  // Código para concatenar las colas
  finC1->siguiente = inicioC2;

  // Desplegar los elementos concatenados
  q = inicioC1;
  cout << "\n Los elementos concatenados son: \n";
  while ( q != NULL ) {
    cout << q->dato << endl;
    q = q->siguiente;
  };

  return 0;
}
```

Ejercicio 6.8.- Modificar el programa que construye una cola1 y una cola2, para que determine si las dos colas son iguales o no lo son.

Solución parcial:

```
#include <iostream>
#include <conio.h>
#define TRUE 1
#define FALSE 0


struct s_caja {
  int     dato;
  s_caja *siguiente;
};

int j=0;

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;

  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide el dato
  cout << " Numero entero ? ";
  cin >> q->dato;
  return q;
}

void Formar( s_caja *q,
             s_caja* &caja_inicial,
             s_caja* &caja_final ) {
  if( q == NULL )
    return;
  q->siguiente = NULL;
  // "forma" una caja al final
  if( caja_final == NULL )
    caja_inicial = q;
  else
    caja_final->siguiente = q;
  caja_final = q;
  j++;
}

s_caja *Despachar( s_caja* &caja_inicial, s_caja* &caja_final ) {
  // si ya está vacía
  if( caja_inicial == NULL )
    return NULL;
  s_caja *q;
  q = caja_inicial;
  caja_inicial = caja_inicial->siguiente;
  q->siguiente = NULL;
```

```
  // si queda vacía
  if( caja_inicial == NULL )
    caja_final = NULL;
  return q;
}

int main() {
  s_caja *q;
  s_caja *inicioC1 = NULL, *inicioC2 = NULL, *finC1=NULL,
         *finC2=NULL;
  // crear y guardar elementos de la cola 1
  cout << "Proporciona los elementos de la cola 1 \n";
  while( otra_caja() ) {
    . . .
  };
  cout << " \n La longitud de la cola 1 es: " << j << endl;
  j=0;

  // crear y guardar elementos de la cola 2
  cout << "Proporciona los elementos de la cola 2 \n";
  . . .

  cout << " \n La longitud de la cola 2 es: " << j << endl;

  q = inicioC1;
  cout << "\n Los elementos de la cola 1 son: \n";
  while( q != NULL ) {
    . . .
  };
  q = inicioC2;
  cout << "\n Los elementos de la cola 2 son: \n";
  . . .

  // Determinar si el contenido de la cola 1 y
  // el de la cola 2 es el mismo
  q = inicioC1;
  s_caja *q2 = inicioC2;
  bool iguales;

  if(q == NULL && q2 == NULL)
    // Las dos colas están vacías
    iguales = ...;
  else
  {
   if(q != NULL && q2 != NULL)
      // las dos colas tienen elementos
      iguales = TRUE;
    else
      // una de las dos colas está vacía
      iguales = ...;
  };

  while((...) && q != NULL && q2 != NULL ) {
    iguales = iguales &&(...);
    q = q->siguiente;
    q2 = q2->siguiente;
  };
```

```
  if( iguales )
    cout << . . .;
  else
    cout << . . .;

 return 0;
}
```

Solución completa:

```
#include <iostream>
#include <conio.h>
#define TRUE 1
#define FALSE 0

struct s_caja {
  int     dato;
  s_caja *siguiente;
};

int j=0;

bool otra_caja() {
  char c;
  cout<< "\n Desea introducir los datos de otra caja? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;

  q = new s_caja;
  if( q == NULL )
    return NULL;

  // pide el dato
  cout << " Numero entero ? ";
  cin >> q->dato;

  return q;
}

void Formar( s_caja *q,
             s_caja* &caja_inicial,
             s_caja* &caja_final ) {
  if( q == NULL )
    return;
  q->siguiente = NULL;
  // "forma" una caja al final
  if( caja_final == NULL )
    caja_inicial = q;
  else
    caja_final->siguiente = q;
  caja_final = q;
  j++;
}

s_caja *Despachar( s_caja* &caja_inicial, s_caja* &caja_final ) {
  // si ya está vacía
  if( caja_inicial == NULL )
    return NULL;
  s_caja *q;
  q = caja_inicial;
  caja_inicial = caja_inicial->siguiente;
  q->siguiente = NULL;
```

```
  // si queda vacía
  if( caja_inicial == NULL )
    caja_final = NULL;

  return q;
}

int main() {
  s_caja *q;
  s_caja *inicioC1 = NULL, *inicioC2 = NULL, *finC1=NULL,
         *finC2=NULL;

  // crear y guardar elementos de la cola 1
  cout << "Proporciona los elementos de la cola 1 \n";
  while( otra_caja() ){
    q = hacer_caja( );
    Formar( q, inicioC1, finC1 );
  };

  cout << " \n La longitud de la cola 1 es: " << j << endl;
  j=0;

  // crear y guardar elementos de la cola 2
  cout << "Proporciona los elementos de la cola 2 \n";
  while( otra_caja() ){
    q = hacer_caja( );
    Formar( q, inicioC2, finC2 );
  };

  cout << " \n La longitud de la cola 2 es: " << j << endl;

  q = inicioC1;


  cout << "\n Los elementos de la cola 1 son: \n";
  while ( q != NULL ) {
    cout << q->dato << endl;
    q = q->siguiente;
  };

  q = inicioC2;
  cout << "\n Los elementos de la cola 2 son: \n";
  while ( q != NULL ) {
    cout << q->dato << endl;
    q = q->siguiente;
  };

  // Determinar si el contenido de la cola 1 y
  // el de la cola 2 es el mismo
  q = inicioC1;
  s_caja *q2 = inicioC2;
  bool iguales;

  if(q == NULL && q2 == NULL)
    // Las dos colas están vacías
    iguales = TRUE;
  else
```

```
  {
   if(q != NULL && q2 != NULL)
      // las dos colas tienen elementos
      iguales = TRUE;
    else
      // una de las dos colas está vacía
      iguales = FALSE;
  };

  while((iguales) && q != NULL && q2 != NULL ) {
    iguales = iguales &&(q->dato == q2->dato);
    q = q->siguiente;
    q2 = q2->siguiente;
  };

  if (iguales)
    cout << "El contenido de las dos colas es el mismo";
  else
    cout << "El contenido de las dos colas es diferente";

  return 0;
}
```

Ejercicio 6.9.- El código del *ejercicio 6.8* que compara las dos colas se puede rescribir de una manera más elegante. Completa el siguiente fragmento de código:

```
// Determinar si el contenido de la cola 1 y la cola 2 es el mismo
  q = inicioC1;
  s_caja *q2 = inicioC2;

  while(q != NULL && q2 != NULL && (...) ){
    q = q->siguiente;
    q2 = q2->siguiente;
  };


  if (q == ... && q2 == ...)
    cout << "El contenido de las dos colas es el mismo";
  else
    cout << "El contenido de las dos colas es diferente";

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Solución completa:

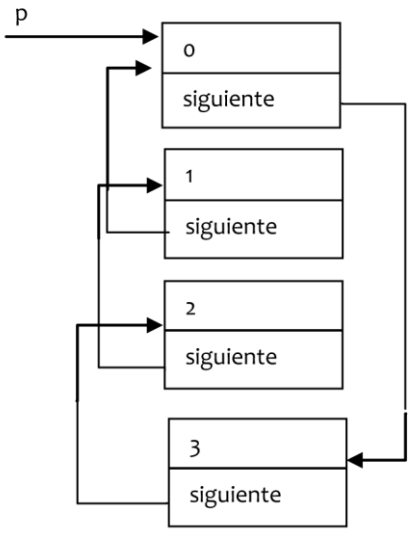
```
// Determinar si el contenido de la cola 1 y la cola 2 es el mismo
  q = inicioC1;
  s_caja *q2 = inicioC2;

  while(q != NULL && q2 != NULL && (q->dato == q2->dato) ){
    q = q->siguiente;
    q2 = q2->siguiente;
  };


  if (q == NULL && q2 == NULL)
    cout << "El contenido de las dos colas es el mismo";
  else
    cout << "El contenido de las dos colas es diferente";

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Ejercicio 6.10.- Dibujar la lista ligada que se genera con el siguiente fragmento de programa:

```
…
struct s_caja {
  int elemento;
  s_caja *siguiente;
  s_caja *anterior;
}

s_caja *p, *q;

p = NULL;
for( int i=0; i<4; i++ ) {
  q = new s_caja;
  q->elemento = i;
  q->siguiente = q;
  if( p == NULL ) p = q;
  q->siguiente = p->siguiente;
  p->siguiente = q;
}
…
```

Solución:

Se genera la siguiente *lista ligada circular*:

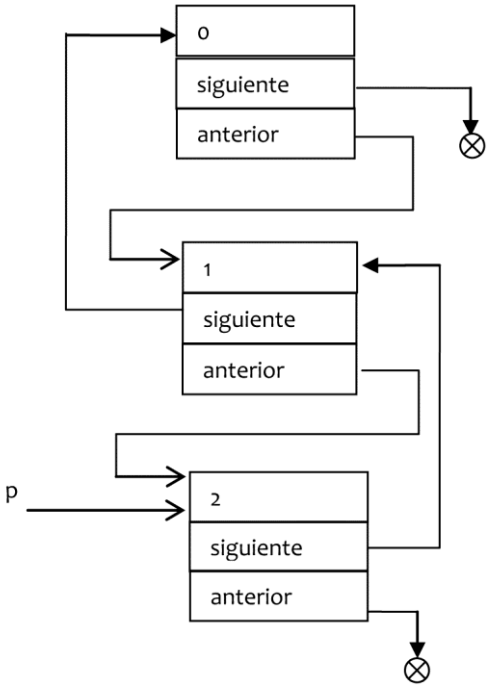


Ejercicio 6.11.- Dibujar la lista ligada que se genera con el siguiente fragmento de programa:

```
…
struct s caja {
  int elemento;
  s_caja *siguiente;
  s_caja *anterior;
}
s_caja *p, *q;
p = NULL;

for( int i=0; i<3; i++ ) {
  q = new s_caja;
  q->elemento = i;
  q->siguiente = p;
  q->anterior  = NULL;
  if( p != NULL )
    p->anterior  = q;
  p = q;
}
…
```

Solución:

Se genera la siguiente *lista doblemente ligada.*



Capítulo VII Algoritmos de Búsqueda y Ordenamiento

Pedro Pablo González Pérez
María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

Comprender los conceptos y elaborar programas que contengan:

- Búsqueda secuencial y binaria
- Métodos directos de ordenamiento

VII.1 Búsqueda en arreglos

La búsqueda en arreglos es el proceso que permite determinar si un arreglo contiene un valor que coincida con un cierto valor llave y consiste en encontrar un elemento específico dentro del arreglo. Las búsquedas secuencial y binaria son los tipos básicos de búsqueda, también existen mecanismos mas sofisticados.

VII.1.1 Búsqueda Secuencial

La búsqueda secuencial se puede aplicar tanto a un arreglo ordenado como a uno no ordenado. Consiste en buscar un elemento en el arreglo a partir de un valor clave especificado. Los elementos del arreglo se exploran en secuencia, uno a continuación del otro, hasta que se encuentre el valor buscado o se llegue al fin del arreglo sin encontrarlo. A lo largo de la búsqueda se compara cada elemento del

arreglo con el elemento que se desea encontrar. Asumiendo que el elemento que se busca existe en el arreglo, el número máximo de iteraciones que se realizan con la búsqueda secuencial para encontrar el elemento clave no es predecible debido a que el arreglo puede estar o no estar ordenado. Por otra parte, el método secuencial no es un método de búsqueda eficiente cuando el arreglo está ordenado.

En la Figura VII-1 se presenta el algoritmo para realizar una búsqueda secuencial usando como ejemplo la búsqueda del valor **15** en un arreglo desordenado con 10 datos.

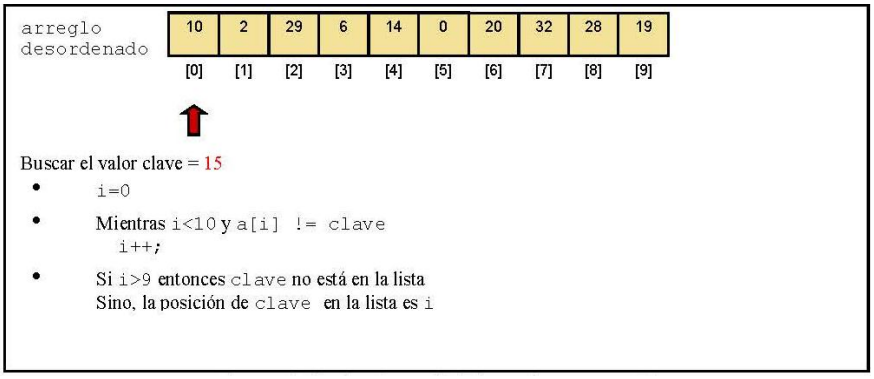


Figura VII-1 Algoritmo de búsqueda secuencial.

Ejemplo 7.1.-

- Escribir un programa que pida al usuario una lista de 20 números y que los almacene en un arreglo.
- Hacer que el programa pida al usuario un número más y que determine si este número está en el arreglo y, de ser así, en qué posición está.
- Realizar lo anterior por medio del algoritmo de la *búsqueda secuencial*.

Solución:

```
//búsqueda secuencial en un arreglo
#include < iostream.h>

int A[20], buscado, i;
main( )
{
   for( i=0; i < 20; i++){
     cout << "A[" << i << "]=?";
     cin >> A[i];
   };
   cout << "Que numero quieres buscar?"; cin >> buscado;
   i = 0;
   while ( i<20 && A[i]!=buscado )
     i++;
   if( i > 19 )
     cout << " El numero buscado no esta en la lista;
   else
     cout << " El numero buscado esta en la posición: "
          << i << " del arreglo";

   return 0;
}
```

La primera condición se evalúa primero y si es falsa ya no se ejecuta la segunda! (Con este orden nunca se busca fuera del arreglo)

VII.1.2 Búsqueda Binaria.

La búsqueda binaria se aplica solamente en arreglos ordenados. Resulta mucho más eficiente que la búsqueda secuencial sin embargo la condición es que el arreglo esté ordenado. El método de búsqueda binaria calcula el índice central del arreglo (valor medio) y ubica en éste el punto de lectura. Si el valor clave es mayor que este punto central, entonces la lectura se efectúa en la mitad superior del arreglo, de lo contrario la lectura se efectúa en la mitad inferior del arreglo. Una vez seleccionada la mitad del arreglo donde podría encontrarse el valor clave, el proceso de encontrar el índice central (valor medio) del sub arreglo se repite.

Considerando que el arreglo `a[]` está ordenado en orden ascendente, entonces denotemos los índices inferior y superior del arreglo por: `Iinf = 0, Isup = n-1`, respectivamente, `n` la cantidad de elementos en el arreglo y `clave` el valor a buscar en él. El algoritmo de búsqueda binaria es el siguiente:

1. Calcular el índice que representa el punto medio del arreglo:
   `Imedio = (Iinf + Isup)/2` (división entera)

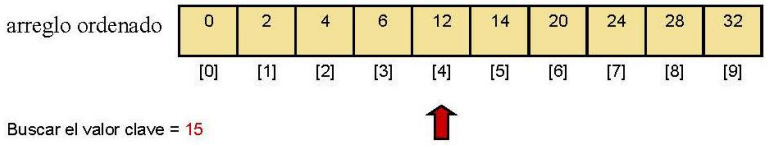
2. Comparar el valor del punto medio con el valor de la clave:
   Si `a[Imedio]<clave`, considerar como nuevo subarreglo de búsqueda aquel con valores extremos `Iinf = Imedio+1` y `Isup = Isup`. Si

`a[Imedio]>clave`, considerar como nuevo subarreglo de búsqueda aquel con valores extremos `Iinf = Iinf` y `Isup = Imedio-1`

3. Una vez seleccionado el nuevo subarreglo, regresar al punto 1.

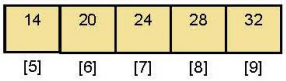
4. Condición de finalización del algoritmo:
   - El valor clave ha sido encontrado.
   - Se llega a un subarreglo de un único elemento (`Iinf = Isup`) que no coincide con el valor clave buscado.

Ejemplo 7.2.- Buscar el valor **15** en el siguiente arreglo de datos ordenados de forma ascendente utilizando la búsqueda binaria.



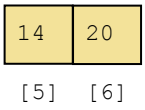
1.- `Imedio = (Iinf + Isup)/2 = (0+9)/2 = 4`

2.- `clave = 15 > a[4] = 12` Por lo tanto, considerar como nuevo subarreglo de búsqueda: `Iinf = Imedio + 1`



3.- `Imedio = (Iinf + Isup)/2 = (5+9)/2 = 7`

Por lo tanto, considerar como nuevo subarreglo de búsqueda `Isup = Imedio-1.`

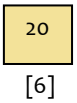


4. `Imedio = (Iinf + lsup)/2 = (5 + 6)/2 = 5`

Por lo tanto, considerar como nueva subarreglo de búsqueda la subarreglo de un único elemento:

```
Iinf = Imedio+1  (ahora linf es igual a Isup)
Imedio = (Iinf + Isup)/2 = (6 + 6)/2 = 6
```

El elemento medio de este subarreglo es el propio elemento `a[6] = 20`, el cual es mayor que el elemento `clave 15`.



Como la búsqueda no puede continuar en una sublista vacía, entonces se concluye que el elemento `clave 15` no se encuentra en el arreglo.

El método de búsqueda binaria es mucho más eficiente que la búsqueda secuencial ya que el número de operaciones necesario se reduce. En la primera iteración se descarta a la mitad del arreglo. Cuando el tamaño del arreglo es más grande, se descartan más elementos con lo que la ventaja sobre el método secuencial es mayor.

El número máximo de iteraciones necesarias k para determinar si una clave está o no en un arreglo de tamaño n está dado por k=log2(n+1) que es muy bueno comparado con la búsqueda secuencial en la que k=n.

Ejemplo 7. 3.-

- Escribir un programa que pida al usuario una lista de 20 números en orden ascendente,
- que pida al usuario un número y que determine si el número está en la lista y en qué posición.
- Para lograr lo anterior usar *búsqueda binaria*.

Solución A:

```
//búsqueda binaria en un arreglo Solución A
#include < iostream.h>

int A[20], buscado, i, linf, lsup, mitad;

main( ) {

  for( i=0; i < 20; i++){
    cout << "A[" << i << "]=?";
    cin >> A[i];
  };
  cout << "Que numero quieres buscar?"; cin >> buscado;
  linf = 0; lsup = 19;

  do {
    mitad = (linf + lsup)/2;              // división entera
    if (A[mitad] < buscado)
      linf = mitad +1;
    else
      lsup = mitad -1;
  }while(A[mitad] != buscado && linf <= lsup);

  if( A[mitad] == buscado )
    cout <<"El numero está en la posición:"<< mitad <<"de A";
  else
    cout << "Lastima!! el numero no esta…";

  return 0;
}
```

En la programación existen diferentes formas de lograr un mismo objetivo, una muestra de esto son las siguientes tres soluciones alternativas a la *solución A*.

Solución B:

```
//búsqueda binaria en un arreglo Solución B
#include < iostream.h>

int A[20], buscado, i, linf, lsup, mitad;

main( ) {

  for( i=0; i < 20; i++){
    cout << "A[" << i << "]=?";
    cin >> A[i];
  };
  cout << "Que numero quieres buscar?"; cin >> buscado;
  linf = 0; lsup = 19;

  do {
    mitad = (linf + lsup)/2;              // división entera
    if( A[mitad] < buscado ) linf = mitad +1;
    if( A[mitad] > buscado ) lsup = mitad -1;
  } while( A[mitad] != buscado && linf <= lsup );

  if( A[mitad] == buscado )
    cout << "El numero está en la posición: "
         <<mitad<<" de A";
  else
    cout << "Lastima!! el numero no esta…";

  return 0;
}
```

Solución C:

```
//búsqueda binaria en un arreglo Solución C
#include < iostream.h>

int A[20], buscado, i, linf, lsup, mitad, encontrado = 0;

main( ) {

  for( i=0; i < 20; i++){
    cout << "A[" << i << "]=?";
    cin >> A[i];
  };
  cout << "Qué número quieres buscar?"; cin >> buscado;
  linf = 0; lsup = 19;

  do {
    mitad = (linf + lsup)/2;              // división entera
    if( A[mitad] == buscado ) encontrado = 1;
    if( A[mitad] < buscado ) linf = mitad +1;
    if( A[mitad] > buscado ) lsup = mitad -1;
  } while( encontrado != buscado && linf <= lsup );

  if( A[mitad] == buscado )
    cout << "El número está en la posición: "<<mitad<<"de A";
  else
    cout << "Lastima!! el numero no esta…";

  return 0;
}
```

A esto se le llama bandera

Solución D:

```
//búsqueda binaria en un arreglo Solución D
#include < iostream.h>

int A[20], buscado, i, linf, lsup, mitad;

main( ) {

  for( i=0; i < 20; i++){
    cout << "A[" << i << "]=?";
    cin >> A[i];
  };
  cout << "Qué número quieres buscar?"; cin >> buscado;
  linf = 0; lsup = 19;

  while( true ) {
    mitad = ( linf + lsup )/2;
    if( A[mitad] == buscado || linf == lsup )
      break;
    if( A[mitad] < buscado ) linf = mitad +1;
    if( A[mitad] > buscado ) lsup = mitad -1;
  };

  if( A[mitad] == buscado )
    cout << "El número está en la posición: "<<mitad<<"de A";
  else
    cout << "Lástima!! el número no está…";

  return 0;
}
```

VII.1.3 Ejercicios de búsqueda.

Ejercicio 7.1.- Completar el siguiente programa para que pida al usuario el tamaño de un arreglo, que lo aloje en la memoria dinámica y que pida sus datos. Solicitar al usuario un número a buscar dentro del arreglo dado y que el programa determine si este número está en el arreglo y, de ser así, en qué posición está.

```
#include <iostream>

int main( ) {

  //búsqueda secuencial en un arreglo
  int ..., n, buscado;
  cout << "¿ De qué tamaño es el arreglo A? ";
  cin >> ...;

  //Alojar el arreglo A
  . . .

  // ¿Cuáles son los datos del arreglo A ?
  for(int i=0; i < n; i++) {
    cout << "A[" << . . . << "]= ";
    cin >> . . .;
  };

  cout << "Qué número quieres buscar? ";
  cin >> . . .;
  int i = 0;
  while ( . . . )
     i++;
  if( i > n )
    cout << " El número buscado no está en la lista";
  else
    cout << " El número buscado está en la posición: "
         << i << " del arreglo";

  delete [ ] A;

  return 0;
}
```

Solución:

```
#include <iostream>

int main( ) {

  //búsqueda secuencial en un arreglo
  int *A, n, buscado;
  cout << "¿ De qué tamaño es el arreglo A? ";
  cin >> n;

  //Alojar el arreglo A
  A = new int[n];

  // ¿Cuáles son los datos del arreglo A ?
  for(int i=0; i < n; i++) {
    cout << "A[" << i << "]= ";
    cin >> A[i];
  };

  cout << "Qué número quieres buscar? "; cin >> buscado;
  int i = 0;
  while ( i < n && A[i] != buscado )
    i++;
  if( i > n )
    cout << " El número buscado no está en la lista";
  else
    cout << " El número buscado está en la posición: "
         << i << " del arreglo";

  delete [ ] A;

  return 0;
}
```

Ejercicio 7.2.- Hacer la función:

```
busquedasecuencial(int elemento, int *arreglo, int n)
```

La función debe regresar la posición índice del elemento si se encuentra en el arreglo y `-1` si no lo encuentra en el arreglo. Repetir el ejercicio 1 empleando esta función.

Solución:

```
#include <iostream>

int busquedasecuencial(int elemento, int *arreglo, int n){

  int i = 0;
  while ( i < n && arreglo[i] != elemento )
    i++;

  if( i>=n )
    return  -1;
  else
    return i;
}

int main( ) {

  //búsqueda secuencial en un arreglo
  int n;
  int *A, buscado;
  cout << "¿ De que tamaño es el arreglo A? ";
  cin >> n;

  //Alojar el arreglo A
  A = new int[n];

  // ¿Cuales son los datos del arreglo A ?
  for(int i=0; i < n; i++) {
    cout << "A[" << i << "]= ";
    cin >> A[i];
  };

  cout << "Que numero quieres buscar? "; cin >> buscado;
  int i ;
  i = busquedasecuencial(buscado, A, n);

  if( i < 0 )
    cout << " El numero buscado no esta en la lista";
  else
    cout << " El numero buscado esta en la posición: "
         << i << " del arreglo";

  delete [ ] A;

  return 0;
}
```

Ejercicio 7.3.- Completar el programa para que lea números enteros de un archivo y que los almacene en un arreglo de tamaño variable. El primer dato que lee del archivo es el número de datos guardados en éste. Preguntar al usuario por un número y determinar si este número está en el arreglo usando búsqueda binaria.

```
#include <iostream>
#include <. . .>

int main( ) {

  int tamanio, linf, lsup, buscado, mitad;
  int *A;

  . . . entrada;
  entrada...("ArregloOrdenado.txt");
  if( !entrada.is_open() ) {
    cout << "no se pudo abrir el archivo" << endl;
    system("PAUSE");
    return -2;
  };

  entrada >> tamanio;
  cout << "El arreglo tiene " << . . .
       << " elementos y son: \n";

  // Alojar el arreglo A

  // Leer los datos del archivo
  for( ) {
    . . .
    cout << "A[" << i << "]= "<< . . . << endl;
  };

  cout << "Que numero quieres buscar? ";
  cin >> buscado;
  linf = . . .;
  lsup = . . .;

  while( true ) {
    mitad = ( linf + lsup )/2;
    if( A[mitad] . . . buscado . . . linf == lsup )
      break;
    if( A[mitad] . . . buscado ) linf = mitad +1;
    if( A[mitad] . . . buscado ) lsup = mitad -1;
  };

  if( A[mitad] . . . buscado )
    cout << "El numero buscado esta en la posicion: "
         << . . . << " de A \n";
  else
    cout << "Lastima!! el numero no esta. . .\n";

  return 0;
}
```

Solución:

```
#include <iostream>
#include <fstream>

int main( ) {

  int tamanio, linf, lsup, buscado, mitad;
  int *A;

  ifstream entrada;
  entrada.open("ArregloOrdenado.txt");
  if( !entrada.is_open() ) {
    cout << "no se pudo abrir el archivo" << endl;
    system("PAUSE");
    return -2;
  };

  entrada >> tamanio;
  cout << "El arreglo tiene " << tamanio
       << " elementos y son: \n";

  //Alojar el arreglo A
  A = new int[tamanio];

  // Leer los datos del archivo
  for(int i=0 ; i<tamanio; i++) {
    entrada >> A[i];
    cout << "A[" << i << "]= "<< A[i] << endl;
  };

  cout << "Que numero quieres buscar? "; cin >> buscado;
  linf = 0;
  lsup = tamanio-1;

  while( true ) {
    mitad = ( linf + lsup )/2;
    if( A[mitad] == buscado || linf == lsup )
      break;
    if( A[mitad] < buscado ) linf = mitad +1;
    if( A[mitad] > buscado ) lsup = mitad -1;
  };

  if( A[mitad] == buscado )
    cout << "El numero buscado esta en la posicion: "
         << mitad << " de A \n";
  else
    cout << "Lastima!! el numero no esta. . .\n";

  return 0;
}
```

Ejercicio 7.4.- Hacer una función con el siguiente prototipo:

```
int busquedabinaria(int elemento, int *arreglo, int n);
```

La función debe regresar la posición índice del elemento si se encuentra en el arreglo y, si no se encuentra en el arreglo, regresar un -1. Repetir el ejercicio 3 empleando esta función.

Solución:

```
#include <iostream>
#include <fstream>

int busquedabinaria(int elemento, int *arreglo, int n) {

  int linf=0, lsup, mitad, encontrado = -1;

  lsup = n-1;

  do {
    mitad = (linf + lsup)/2;     // división entera
    if( arreglo[mitad] == elemento ) encontrado = 1;
    if( arreglo[mitad] < elemento ) linf = mitad +1;
    if( arreglo[mitad] > elemento ) lsup = mitad -1;
  } while( (encontrado != 1) && linf <= lsup );

  if( encontrado == 1 )
    return mitad;
  else
    return -1;
}

int main( ) {

  int tamanio, resultado, buscado;
  int *A;

  ifstream entrada;
  entrada.open("ArregloOrdenado.txt");
  if( !entrada.is_open() ) {
    cout << "no se pudo abrir el archivo" << endl;
    system("PAUSE");
    return -2;
  };

  entrada >> tamanio;
  cout << "El arreglo tiene " << tamanio
       << " elementos y son: \n";

  A = new int[tamanio];
  for( int i=0 ; i<tamanio; i++ ){
    entrada >> A[i];
    cout << "A[" << i << "]= "<< A[i] << endl;
  };

  cout << "Que numero quieres buscar? "; cin >> buscado;
  resultado = busquedabinaria(buscado, A, tamanio);

  if( resultado == -1 )
    cout << "Lastima!! el numero no esta. . .\n";
  else
    cout << "El numero buscado esta en la posicion: "
         << resultado << " de A \n";

  return 0;
}
```

VII.2 Ordenamiento

El ordenamiento o clasificación de datos se refiere a la disposición de un conjunto (estructura) de datos en algún orden determinado con respecto a uno de los campos de los elementos del conjunto. Es decir, el ordenamiento es el problema que se presenta cuando tenemos un conjunto de *n* elementos: a_1, a_2,…, a_n y queremos obtener el mismo conjunto pero de tal forma que:

$$a_i \leq a_{i+1} \qquad i = 1, 2, ..., n$$

O bien que:

$$a_i \geq a_{i+1} \qquad i = 1, 2, ..., n$$

En la vida real se presentan conjuntos de datos que pueden ser bastante complejos. Sin embargo estos problemas de ordenamiento se pueden reducir al ordenamiento de números enteros si utilizamos las claves de los registros o bien índices. Por lo anterior es importante estudiar el problema de ordenamiento de números enteros. Existen numerosos algoritmos para resolver el problema de ordenamiento y podemos clasificarlos en *directos, indirectos* y *otros*.

Entre los algoritmos de ordenamiento *directos* tenemos:

- Ordenamiento por intercambio
- Ordenamiento por burbuja
- Ordenamiento por inserción
- Ordenamiento por selección

Entre los algoritmos de ordenamiento *indirectos* tenemos:

- Ordenamiento por mezcla (Mergesort)
- Ordenamiento rápido (Quicksort)
- Ordenamiento por Montículos (Heap sort)

También existen *otros* algoritmos de ordenamiento, como por ejemplo:

- Ordenamiento por Incrementos (Shell sort)
- Ordenamiento por Cubetas (Bin sort)
- Ordenamiento por Resíduos (RadVIII)

En algunos casos de la práctica es necesario que los datos estén colocados en un orden determinado, ya sea de mayor a menor, de menor a mayor o en orden alfabético. Por ejemplo, recordar que antes de realizar la búsqueda binaria es necesario que el arreglo esté ordenado de menor a mayor.

Se pueden ordenar los elementos de un arreglo, de una lista ligada o de un arreglo de registros, en los últimos dos casos, es necesario elegir un campo con respecto al cual se ordenarán los registros.

Por ejemplo, si tenemos el arreglo `empleados` con la siguiente estructura:

```
struct s_persona{
  char nombre[25];
  int  edad;
  float salario;
};
s_persona empleados[100];
```

Podemos ordenar el arreglo en función del `salario`, o también lo podemos ordenar en función de la `edad`, incluso podría ordenarse en función del `nombre` respetando el orden alfabético.

Si en lugar del arreglo anterior ahora tenemos el arreglo `resultadosExamen` con la siguiente estructura:

```
struct s_Examen{
  long double matricula[10];
  int calificacion;
};

s_examen resultadosExamen[60];
```

Podríamos ordenar el arreglo en función de la `calificacion`, o también en función del número de `matricula`.

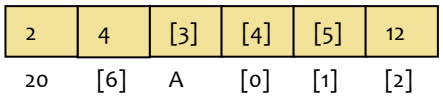
En este capítulo estudiaremos; los métodos directos. En el capítulo de "Recursión" se estudiarán los métodos de ordenamiento rápido y por mezcla.

VII.2.1 El método de "Intercambio"

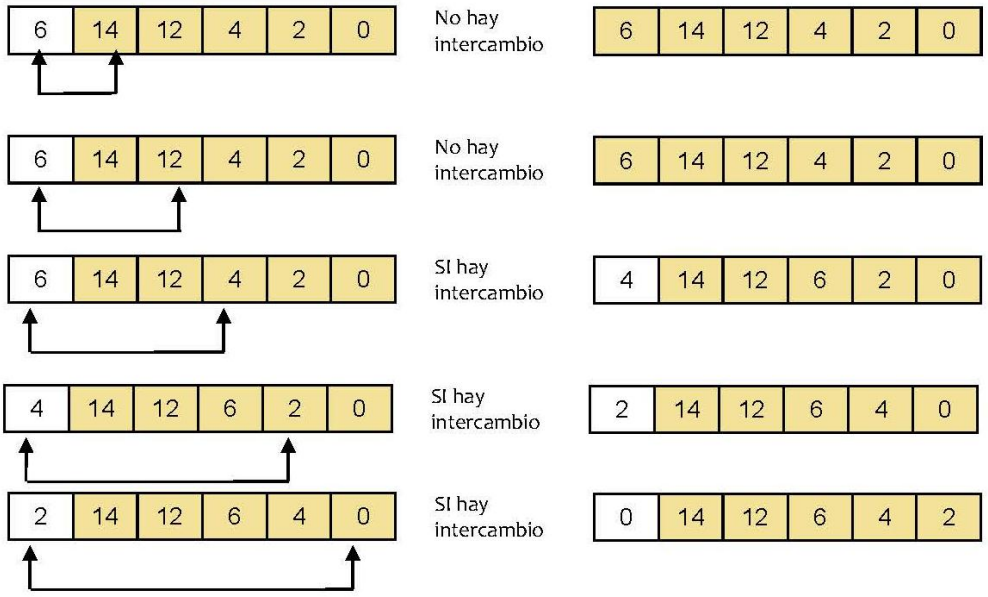
El algoritmo de *ordenamiento por intercambio* es uno de los mas sencillos, ordena los datos en forma ascendente, es decir, de menor a mayor. Este algoritmo se basa en leer varias veces la lista a ordenar, a cada una de estas lecturas se le llama "pase" o "pasada".

El proceso de ordenamiento inicia tomando como referencia el valor del elemento en la posición [0] de la lista con los elementos restantes (posiciones [1], [i], [i+1], …,[n]), se efectúa un intercambio de valores entre las posiciones [0] e [i] (i = 1, 2, …, n) cuando el elemento que esta en la posición [i] sea menor que el elemento de la posición [0]. A todo este proceso se le llama "primera pasada". En la "segunda pasada" se realiza el mismo proceso descrito anteriormente pero ahora tomando como referencia el elemento de la posición [1] y empezando la comparación desde i = 2, 3,…,n. El proceso continúa hasta llegar a la pasada n-1, donde n es la cantidad de elementos en la lista.

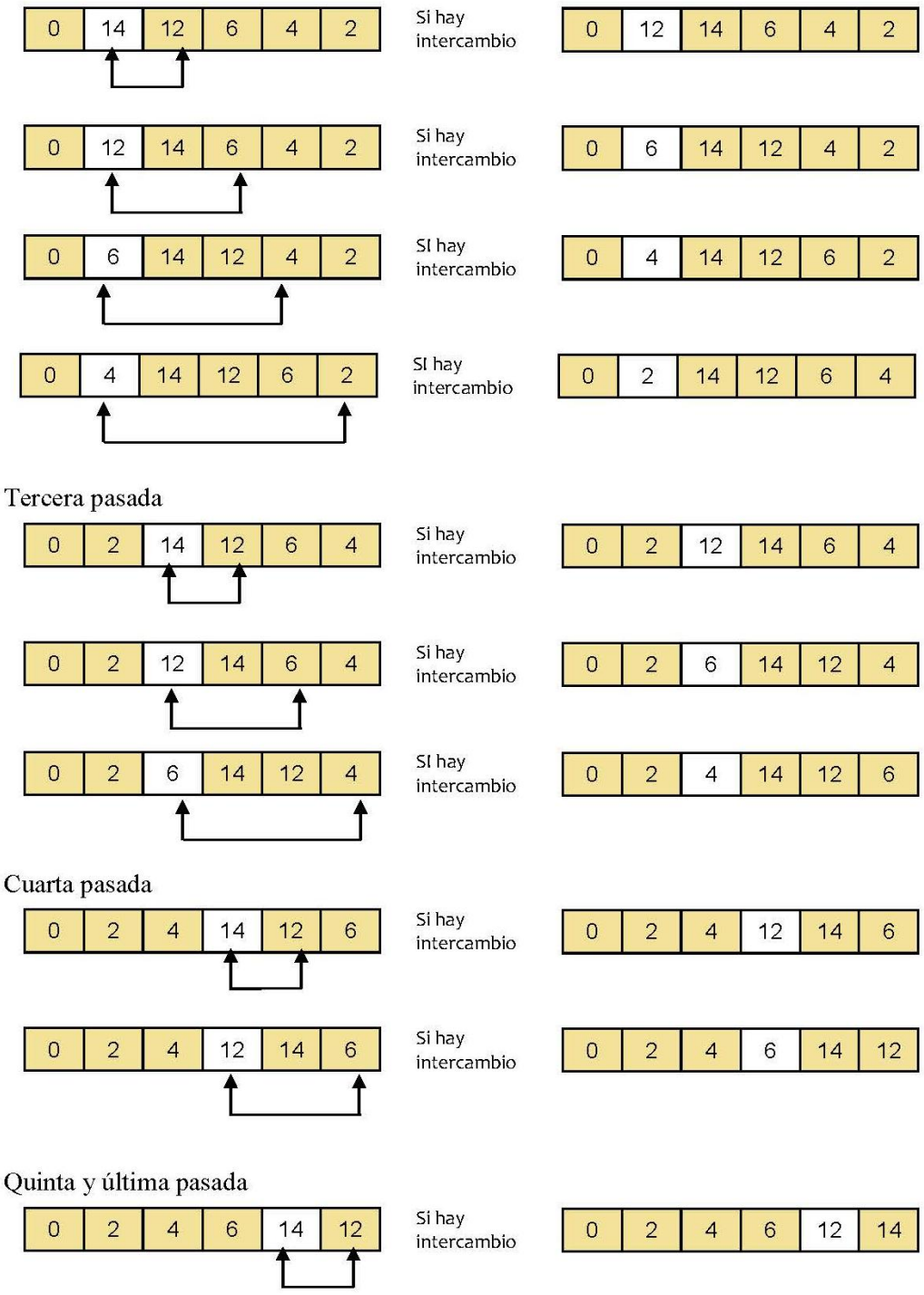
Ejemplo 7.4.- A continuación se presenta como funciona el método de *ordenamiento por intercambio,* cuando se desea ordenar el siguiente arreglo:



Primera pasada



Segunda pasada



Ejercicio 7.5.- Hacer una función en C++ que ordene un arreglo por el método de intercambio. Considerar lo siguiente:

`metodoIntercambio():`

- Recibe como parámetros una lista de elementos y un entero que corresponde a la dimensión de la lista.
- Ejecuta n-1 pasadas para ordenar la lista en orden ascendente.

- Considar dos ciclos for anidados.

Función que ordena un arreglo por el método de Selección:

```
void intercambio( int *a, int n ) {

  int temp;

  for( int i= 0; i < n-1; i++ ){
    for( int j = i+1; j < n; j++ ){
      if( a[i] > a[j] ){
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
      };
    };
  };
}
```

VII.2.2 El método de la "Burbuja".

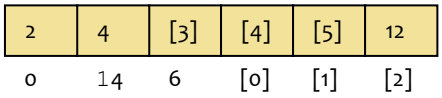
A pesar de que el método de la burbuja es uno de los algoritmos de ordenamiento más conocidos y populares por su facilidad de comprensión e implementación, es a la vez uno de los menos eficientes, de ahí que no resulte muy utilizado en la práctica. El nombre ordenamiento por burbuja se deriva del hecho de que los valores más pequeños en el arreglo flotan o suben hacia la parte inicial (primeras posiciones) del arreglo, mientras que los valores más grandes caen hacia la parte final (últimas posiciones) del arreglo. El método de la burbuja siempre ordena los datos en forma ascendente, es decir, de menor a mayor.

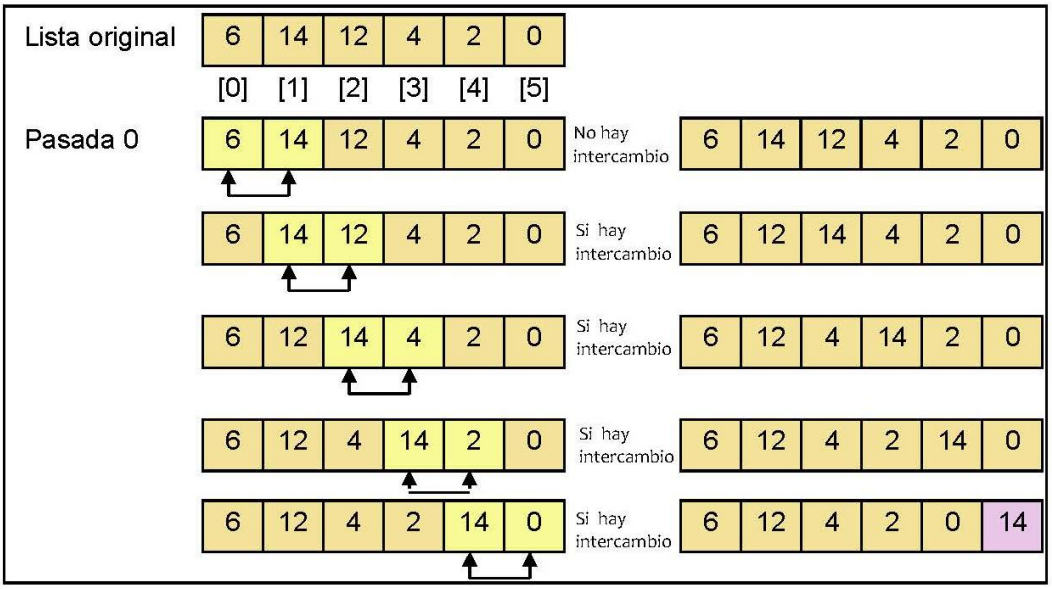
Algoritmo.

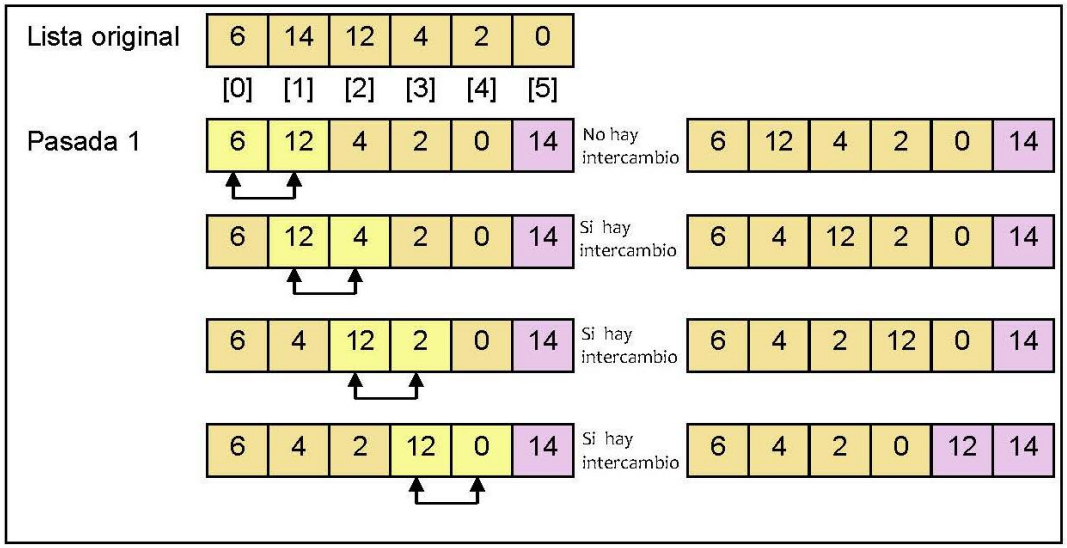
- Una pareja de elementos se define como dos valores en posiciones sucesivas en el arreglo.
- En cada pasada se comparan parejas de elementos.
- Si una pareja está ya ordenada en orden ascendente o si los valores son iguales, entonces se dejan los valores como están.
- Si una pareja está en orden descendente, entonces se intercambian los valores en el arreglo, es decir, si a[i] > a[i+1] entonces temp = a[i], a[i] = a[i+1], a[i+1] = temp.
- Si la dimensión del arreglo es n, entonces el algoritmo requiere n-1 pasadas.
- La formación de parejas se efectúa de la siguiente forma: (a[0], a[1]), (a[1], a[2]), (a[2], a[3]), … (a[i-1], a[i]), (a[i], a[i+1]), …, (a[n-2], a[n-1])
- Al final de la pasada 0, el valor mayor de la lista quedará fijo en la posición [n-1].
- Al final de la pasada 1, el segundo valor mayor de la lista quedará fijo en la posición [n-2] y así sucesivamente.
- De esta forma, la primera parte de la lista en quedar ordenada en orden creciente será la última, mientras que los valores más pequeños continuarán flotando hasta que el algoritmo haya ejecutado las n-1 pasadas
- El ordenamiento concluye con la pasada n-1 con la cual el valor menor se coloca en la posición [0].

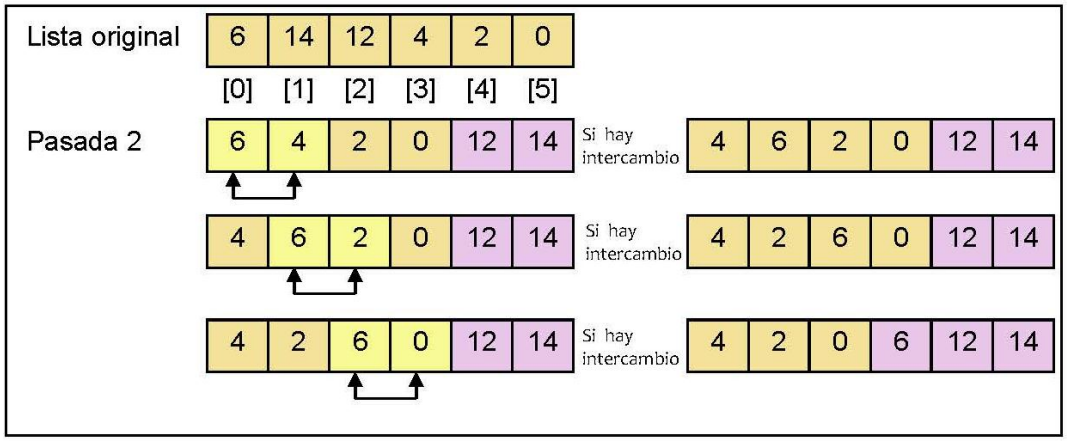
Ejemplo 7.5.-

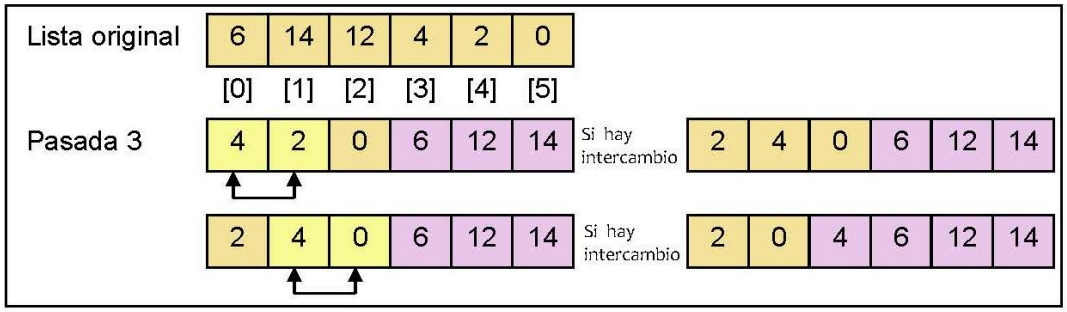
A continuación se presenta como funciona el método de *ordenamiento por el método de la burbuja*, cuando se desea ordenar el siguiente arreglo:

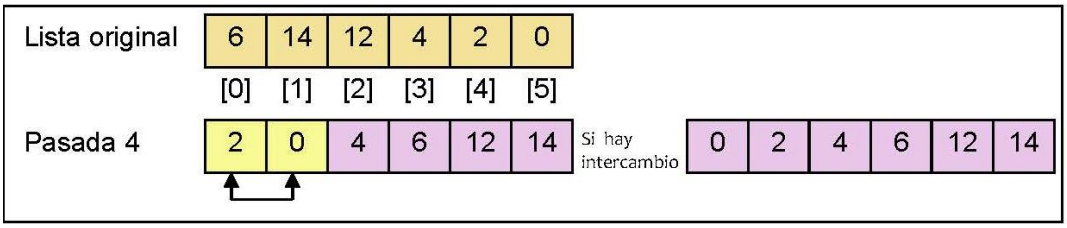





Lista original
6 14 12 4 2 0
[0] [1] [2] [3] [4] [5]
Pasada 1
6 12 4 2 0 14 No hay intercambio 6 12 4 2 0 14
6 12 4 2 0 14 Si hay intercambio 6 4 12 2 0 14
6 4 12 2 0 14 Si hay intercambio 6 4 2 12 0 14
6 4 2 12 0 14 Si hay intercambio 6 4 2 0 12 14


Lista original
6 14 12 4 2 0
[0] [1] [2] [3] [4] [5]
Pasada 2
6 4 2 0 12 14 Si hay intercambio 4 6 2 0 12 14
4 6 2 0 12 14 Si hay intercambio 4 2 6 0 12 14
4 2 6 0 12 14 Si hay intercambio 4 2 0 6 12 14


Lista original
6 14 12 4 2 0
[0] [1] [2] [3] [4] [5]
Pasada 3
4 2 0 6 12 14 Si hay intercambio 2 4 0 6 12 14
2 4 0 6 12 14 Si hay intercambio 2 0 4 6 12 14


Lista original
6 14 12 4 2 0
[0] [1] [2] [3] [4] [5]
Pasada 4
2 0 4 6 12 14 Si hay intercambio 0 2 4 6 12 14

Ejercicio 7.6.- Hacer una función en C++ que ordene un arreglo por medio del método de la burbuja. Considerar lo siguiente:

`burbuja()`:

- Recibe como parámetros una lista de elementos y un entero que corresponde a la dimensión de la lista.
- Ejecuta `n-1` pasadas para ordenar la lista en orden ascendente.
- Considerar un ciclo externo `for` para ejecutar `n-1` pasadas en la lista, y otro ciclo `for` anidado para intercambiar los elementos de la lista.

Función que ordena un arreglo por el método de la burbuja

```
void burbuja( int *a, int n ) {

  int temp;

  for( int i= 0; i < n-1; i++ ){
    for( int j = 0; j < n-1-i; j++ ){
      if( a[j] > a[j+1] ){
        temp = a[j];
        a[j] = a[j+1];
        a[j+1] = temp;
      };
    };
  };
}
```

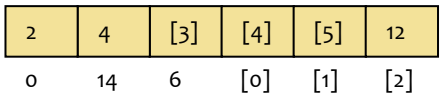
VII.2.3 El método de "Inserción"

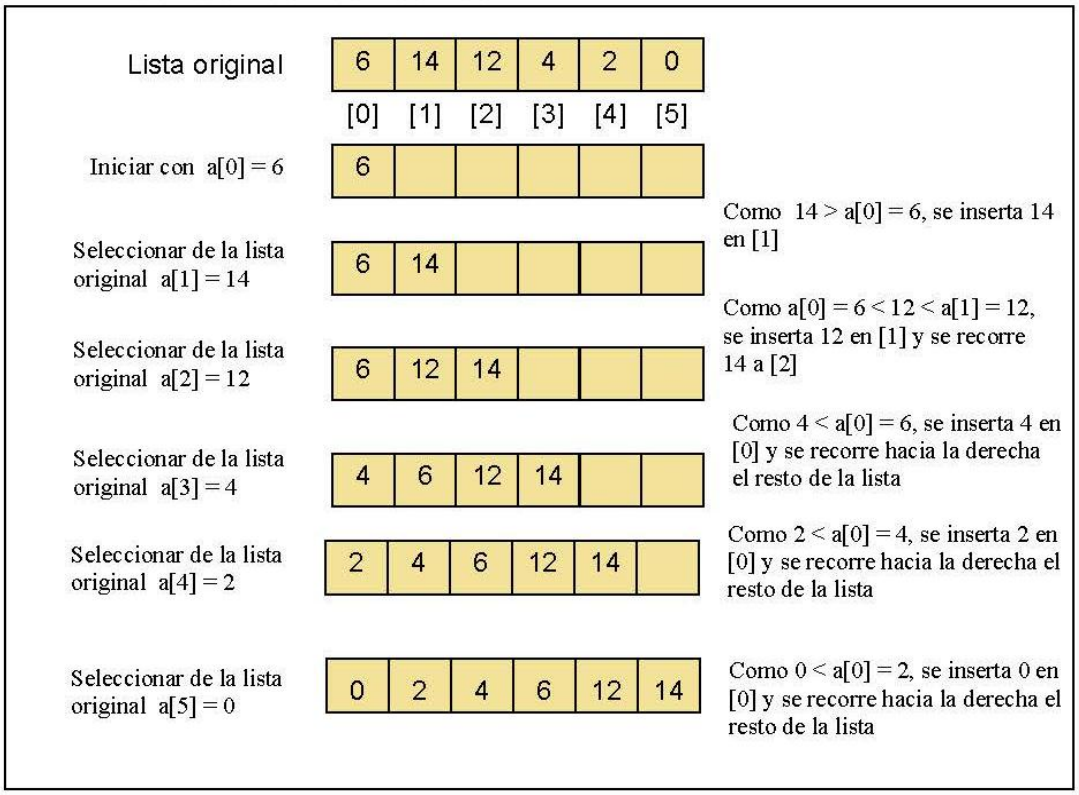
El algoritmo de ordenamiento por inserción directa ordena un arreglo de *n* elementos en orden ascendente, insertando directamente cada elemento de la lista en la posición adecuada y recorriendo hacia la derecha (de [i] a [i+1]) los elementos restantes de la lista que sean mayores al elemento insertado.

Algoritmo

- Se comienza considerando una lista inicial compuesta por un solo elemento: `a[0]`
- A continuación se selecciona `a[1]` y si éste es menor que `a[0]`, entonces `a[1]` se inserta en la posición `[0]` y este último se recorre una posición hacia la derecha, de lo contrario `a[1]` se inserta en la posición `[1]`.
- El proceso continúa seleccionando consecutivamente cada elemento de la sublista `a[i]`, `a[i+1]`, ..., `a[n-1]`, buscando la posición correcta para su inserción en la sublista `a[i-1]`, `a[1]`, ..., `a[0]` y recorriendo hacia la derecha una posición todos los elementos mayores al elemento insertado.

Ejemplo 7.6.- A continuación se presenta como funciona el método de *ordenamiento por el método de inserción directa*, cuando se desea ordenar el siguiente arreglo:





Ejercicio 7.7.- Hacer una función en C++ que ordene un arreglo por medio del método de inserción directa. Considerar lo siguiente:

`insercion():`

- Recibe como parámetros una lista de elementos y un entero que corresponde a la dimensión de la lista.
- Considerar un ciclo externo `for` donde el contador `i` recorre los valores desde `i = 1` hasta `i < n`.
- Se requiere de otro ciclo anidado al `for` para explorar la sublista `a[i-1], a[1], ..., a[0]`

Función que ordena un arreglo por el método de inserción

```
void insercion( int *a, int n ) {

  int j, temp;

  for( int i= 1; i < n; i++ ) {
    temp = a[i];
    for( j = i-1; j>=0 && temp<a[j]; j-- )
      a[j+1] = a[j];

    a[j+1] = temp;
  };
}
```

VII.2.4 El método de "Selección"

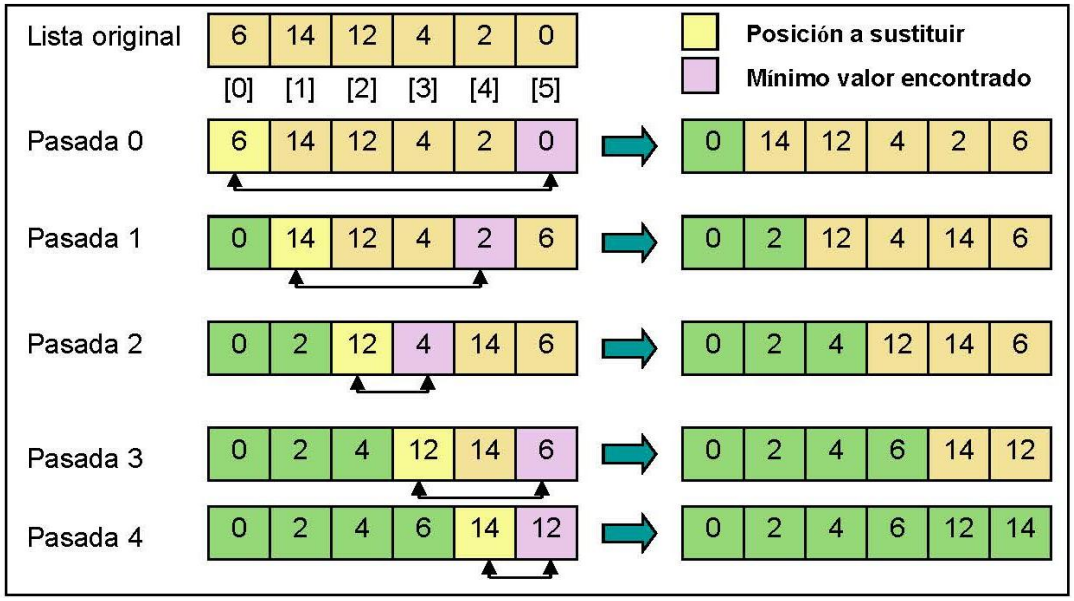
El algoritmo de ordenamiento por selección ordena un arreglo de *n* elementos en forma ascendente. Es decir, si denotamos por `a` el arreglo a ordenar, entonces el resultado debe ser: `a[0] < a[1] < a[2] < ... < a[i] < ... < a[n-1].`

Algoritmo

- Se ejecutan pasadas sucesivas y en cada una de ellas se reemplaza el elemento en la posición a sustituir en la lista por el elemento mínimo encontrado en la lista que se analiza
- La primera posición a sustituir es `a[0]` y la lista que se analiza se forma con `a[0], a[1], a[2], a[i], a[n-1]`
- La segunda posición a sustituir es `a[1]`, y la lista que se analiza se forma con `a[1], a[2], a[3], a[i], a[n-1]`, y así sucesivamente
- En una lista de n elementos se ejecutan n-1 pasadas.

Ejemplo 7.7.- A continuación se presenta como funciona el método de *ordenamiento por el método de selección directa*, cuando se desea ordenar el siguiente arreglo:

2	4	[3]	[4]	[5]	12
0	14	6	[0]	[1]	[2]



Ejercicio 7.8.- Hacer una función en C++ que ordene un arreglo por medio del método de la burbuja. Considerar lo siguiente:

`seleccion():`

- Recibe como parámetros una lista de elementos y un entero que corresponde a la dimensión de la lista.
- Ejecuta `n-1` pasadas para ordenar la lista en orden ascendente.
- Considerar dos ciclos for anidados.
- En el primer ciclo, el contador i recorre los valores desde `i = 0` hasta `i < n-1`.
- En el segundo ciclo, el contador j recorre los valores desde `j = i+1` hasta `j < n`.

Función que ordena un arreglo por el método de Selección:

```
void seleccion( int * a, int n ) {

  int menor, temp, indice;

  for( int i= 0; i < n-1; i++){

    // determinar cual es el menor a partir de i
    menor = a[i];
    indice = i;
    for( int j = i+1; j < n; j++ ){
      if( menor > a[j]){
        menor = a[j];     // el valor del menor
        indice = j;       // la posición del menor
      };
    };

    // poner el menor en la posición i y
    // el de la posición i en la posición del menor
    temp = a[i];
    a[i] = menor;
    a[indice] = temp;
  };
}
```

A continuación presentamos el código del programa principal, este programa contiene un menú para que el usuario elija el método de ordenamiento.

```
main() {

  int opcion,n, *A;
  char c;

  do {
    cout << "Cuantos numeros (enteros) deseas ordenar?"; cin>>n;
    // Alojar el arreglo de tamaño n;
    A = new int[n];

    pedirDatosArreglo(A,n);

    do {
      cout << endl
           <<"elige el tipo de ordenamiento ascendente: \n"
           << " 1 : intercambio " <<endl
           << " 2 : seleccion"    <<endl
           << " 3 : insercion "   <<endl
           << " 4 : burbuja "     <<endl;
      cin >> opcion;
    }while( ( opcion < 1 )||( opcion > 4 ));

    switch(opcion) {
      case 1: intercambio(A,n);
              cout<<"\nLos datos ordenados por intercambio: \n";
              break;
      case 2: seleccion(A,n);
              cout << "\n Los datos ordenados por selección: \n";
              break;
      case 3: insercion(A,n);
              cout << "\n Los datos ordenados por inserción: \n";
              break;
      case 4: burbuja(A,n);
              cout << "\n Los datos ordenados por burbuja: \n";
              break;
    };

    for( int i=0; i<n; i++ )
      cout << "A["<< i << "]=" << A[i] << endl;

    cout<< "oprime ""Esc"" para terminar" <<endl;
    c = getch();

  }while(27 != c);

  cout << " Adios! ";
  return 0;
}
```

Capítulo VIII Algoritmos Recursivos

María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Objetivos

Comprender los conceptos y elaborar programas que contengan:

- Ordenamiento recursivo
- Construcción y recorrido de árboles

VIII.1La Recursión

Recursión o *Recursividad* es una forma de resolver problemás en la cual el problema se divide en partes más pequeñas que (al menos una de estas) se resuelven de la misma manera que el problema original. En programación, la recursividad consiste en *realizar una llamada a una función desde la misma función*.

Figura VIII-1 Ejemplo de recursión.

En la Figura VIII-1 se muestra un ejemplo de recursión: es una imagen de una televisión mostrando una imagen de esa misma televisión mostrando una imagen de esa misma televisión... En otras palabras, una televisión que se contiene a sí misma. Un escritor que escribe la historia de cómo es que él escribió esa historia de cómo es que él escribe la historia... es otro ejemplo de recursión.

En el contexto de la programación, la recursividad es un recurso muy poderoso que ayuda a resolver ciertos problemás de forma más natural, es decir, problemás cuya solución se puede hallar resolviendo el mismo problema pero con un caso de menor tamaño. Existen procesos que son recursivos por naturaleza, como por ejemplo, el factorial, el trabajo con árboles y la obtención de algunas series. Aunque cualquier proceso recursivo se puede representar también como un proceso iterativo, un modelo recursivo en ocasiones resulta más claro.

Podemos ejemplificar con pseudocódigo el aspecto de un algoritmo recursivo de la siguiente manera:

```
ALGORITMO Recursivo(...)
INICIO
 ...
   Recursivo(...);
 ...
FIN
```

El factorial de un número $n \in \mathbf{N}$ es un ejemplo clásico de algoritmo recursivo, y su definición es la siguiente:

$$\text{Factorial } (n) = \begin{cases} 1 & \text{si } n = 0 \\ n*\text{Factorial}(n-1) & \text{si } n > 0 \end{cases}$$

Es decir, el factorial de *n*, se obtiene multiplicando *n* por la función factorial de *n-1* y, cuando *n*= 0 entonces se define que el factorial es 1 lo cual hace que el algoritmo no sea infinito sino que termine. Una codificación en C++ de una función factorial es:

```
long int factorial( long int n ) {
  if( n<=0 )
    return 1;

  return n*factorial( n-1 );
};
```

Obsérvese que la función `factorial` se llama a sí misma.

Para entender el funcionamiento y seguir un proceso recursivo, hay que dividirlo en dos partes:

El proceso de ida (entrada).
El proceso de regreso (salida).

En la Figura VIII-2 se ilustra que la recursión tiene un proceso de ida, en el cual se hacen los llamados recursivos a la función y, posteriormente, cuando se cumple con una condición de terminación de los llamados recursivos, comienza el proceso de regreso en el cual se van entregando soluciones parciales hasta llegar a la función que inició el proceso.

Al número de veces que se ejecuta la función se le llama *profundidad* de la recursión.

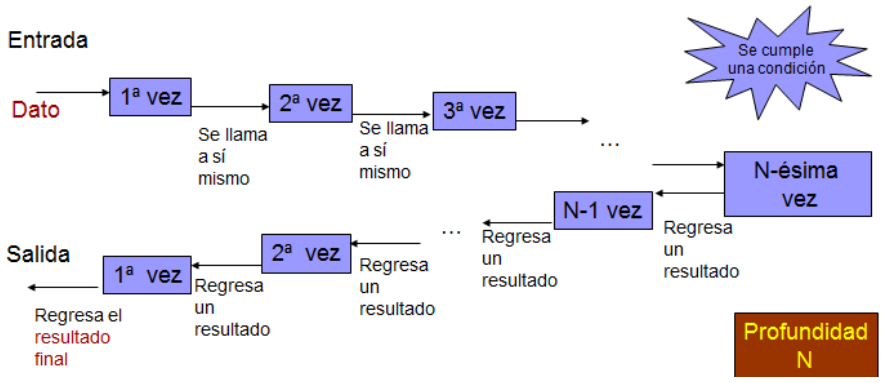


Figura VIII-2 Entrada y salida de la recursión.

Ejemplo 8.1.- como se aprecia en la Figura VIII-3, la profundidad de 5! es 5.

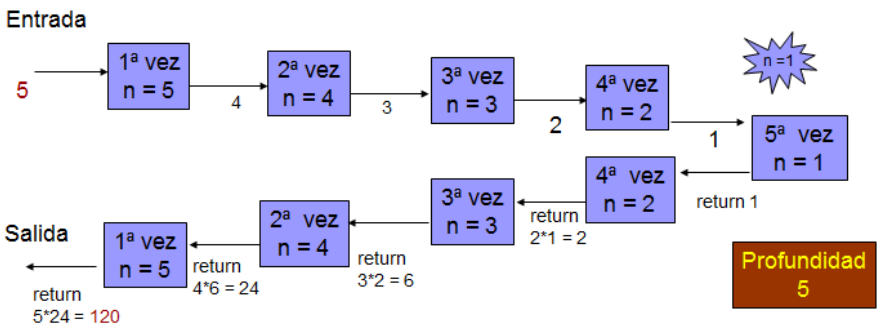


Figura VIII-3 Recursión para obtener 5!

Dada la forma en la que funciona la recursión, para escribir programás recursivos es necesario tomar en cuenta dos aspectos fundamentales:

- La manera de resolver el o los casos base (no recursivos).
- La manera de resolver el caso general en términos de un caso más cercano a los casos base (llamada recursiva).

Ejemplo 8.2.- Para ilustrar el caso base y el caso general veremos el ejemplo de la recursividad en la función de *Fibonacci*.

Para calcular el valor de la función de *Fibonacci* para el n-ésimo elemento de la serie, se tiene lo siguiente:

El número de elementos de la serie es *n* y la serie comienza con el elemento **1**.

- Casos-base: $fib(1) = 1, \quad fib(2) = 1$
- Caso General, para $n>2$: $fib(n) = fib(n-1) + fib(n-2)$

El pseudocódigo de este algoritmo es

```
ALGORITMO Fib( n )
INICIO
  SI (n <= 2) ENTONCES
     DEVOLVER 1
  DEVOLVER = Fib(n-1) + Fib(n-2)
FIN
```

y una de las posibles implementaciones en C++ es:

```
#include <iostream>

int fibonacci( int n ){
  if( n<=2 )
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}

main()
{
  int n;
  cout<< "¿Que elemento de la serie de fibonacci necesitas? \n";
  cin >> n;

  cout << "el elemento " << n << " de la serie es: "
       << fibonacci(n) << endl;
  return 0;
}
```

En la Figura VIII-4 se ilustra la forma en la que se ejecutan los llamados recursivos para obtener el elemento *n*=5 de la serie de *Fibonacci*.

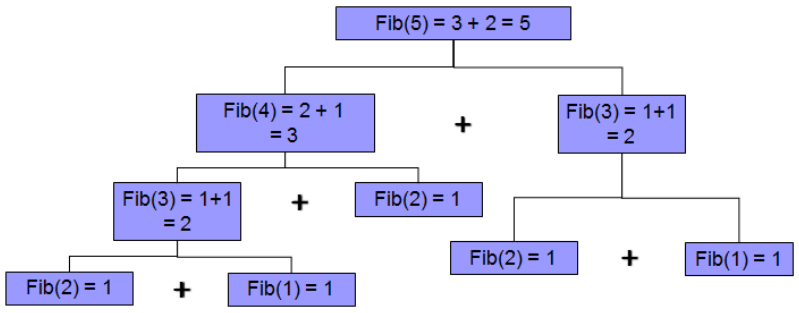


Figura VIII-4 *Recursión para obtener el 5° elemento de la serie de Fibonacci.*

La naturaleza de algunos problemás permite que con la recursión se obtengan algoritmos simples para problemás complejos y, por lo tanto, más claros que los algoritmos iterativos para estos mismos problemás. Sin embargo, las soluciones iterativas son a veces más eficientes debido a que no consumen tanta memoria "stack" de la computadora al hacer los llamados recursivos a las funciones.

VIII.2 Ejercicios de recursividad

Ejercicio 8.1.- Hacer un programa que obtenga el factorial de un número entero. Agregar un `cout` para que se desplieguen los resultados parciales del regreso de la recursión.

Solución parcial:

```
#include <iostream>

long int factorial( long int ... ) {
  long int f=1;
  if( n>0 )
    f = n*factorial( n-1 );
  ...
  return f;
};

main() {
  int n;
  cout << "ingresa un número entero para obtener su factorial ";
  cin >> n;
  cout << "El factorial de " << n << " es: "
       <<  ... << endl;

  return 0;
}
```

Solución completa:

```
#include <iostream>

long int factorial( long int n ) {
  long int f=1;
  if( n>0 )
    f = n*factorial(n-1);
  cout << "regresa " << f << endl;
  return f;
};

main() {
  int n;
  cout<<"ingresa un número entero para obtener su factorial ";
  cin >> n;
  cout << "El factorial de " << n << " es: "
       << factorial(n) << endl;
  return 0;
}
```

Ejercicio 8.2.- Modificar la implementación recursiva del código que obtiene el elemento *n* de la serie de Fibonacci (presentado en la sección anterior) para que se despliegue la serie completa, es decir, hasta el *n*-ésimo elemento, como se muestra en la Figura VIII-5:

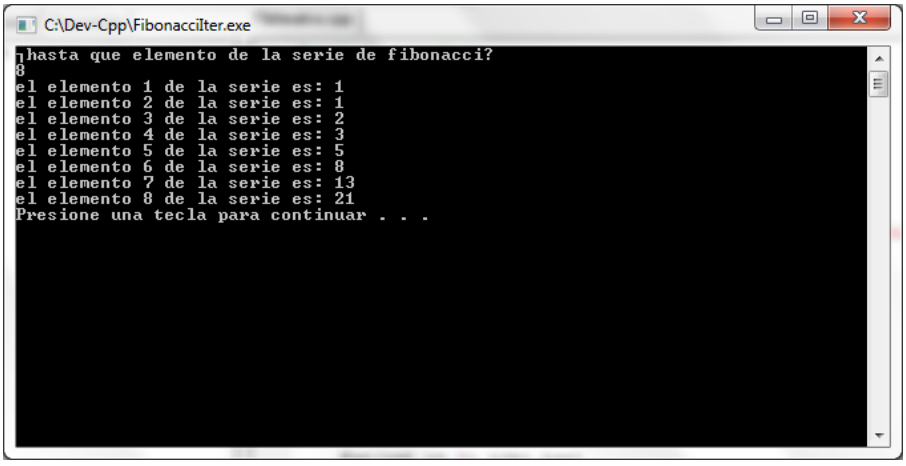


Figura VIII-5 Programa que obtiene los primeros 8 elementos de la serie de Fibonacci.

Solución parcial:

```
#include <iostream>

int fibonacci( int n ) {
  int f=1;
  if( n>2 )
    f = fibonacci( n-1 ) + fibonacci( n-2 );
  ...
  return f;
}

main() {
  int n;
  cout<< "...\n";
  cin >> n;

  ...
  return 0;
}
```

Solución completa:

```
#include <iostream>

int fibonacci( int n ) {
  int f=1;
  if( n>2 )
    f = fibonacci( n-1 ) + fibonacci( n-2 );
  cout << "el elemento " << n << " de la serie es: " << f;
  return f;
}

main() {
  int n;
  cout<< "hasta que elemento de la serie de fibonacci?\n";
  cin >> n;

  fibonacci( n );
  return 0;
}
```

Solución:

```
#include <iostream>

main() {
  int n, *serieFib;
  cout << "¿hasta que elemento de la serie de fibonacci?  \n";
  cin >> n;

  serieFib = new int[n];
  serieFib[0] = 1;
  serieFib[1] = 1;
  if( n>2){
    for( int i=2; i<n; i++ )
      serieFib[i] = serieFib[i-1] + serieFib[i-2];
  };
  for( int i= 0; i<n; i++ )
    cout << "el elemento " << i+1 << " de la serie es: "
         << serieFib[i] << endl;

  return 0;
}
```

¿Qué es más fácil, hacer una implementación iterativa o modificar la recursiva?

Respuesta: desplegar todos los elementos de la serie de Fibonacci se complica con la implementación recursiva. Además, la solución no es eficiente ya que se realizan las mismas operaciones varias veces.

Ejercicio 8.3.- Hacer un programa que permita al usuario construir una ***pila (stack)*** de elementos de tipo:

```
struct s_caja {
  int dato;
  s_caja *enlace;
};
```

El programa deberá contener un menú que pregunte al usuario si desea que se imprima el contenido de la *pila* en orden, o en orden inverso. Usar funciones recursivas.

Solución parcial:

```
#include <iostream>

struct s_caja {
  int    dato;
  s_caja *enlace;
};

void imprimeLista1( . . . ) {
  if( p == NULL ) return;
  imprimeLista1( . . . );
};

void imprimeLista2( . . . ) {
  if( p == NULL ) return;
  imprimeLista2( p->enlace );
};

main() {
  s_caja *p = NULL, *q;
  int elem;
  cout << "Para terminar introduce un número negativo \n";
  do{
    cout << "dame un número entero: ";
    cin>>elem;
    if (elem >= 0){
      q = new s_caja;
      q->dato = elem;
      q->enlace = p;
      p = q;
    };
  }while (elem >= 0);

  int opcion;
  // Recorrido de la lista
  do{
    cout << "Como deseas imprimir la pila? " << endl
         << "  0: en orden (LIFO) \n"
         << "  1: en orden inverso \n";
    cin >> opcion;
  }while( opcion<0 || opcion>1 );

  // Desplegar la lista
  . . .

  // Desalojar la lista
  while( p != NULL ) {
    q = p->enlace;
    delete p;
    p = q;
  };

  return 0;
}
```

Solución completa:

```
#include <iostream>

struct s_caja {
  int    dato;
  s_caja *enlace;
};

void imprimeLista1 (s_caja *p)
{
  if( p == NULL ) return;

  cout << p->dato << endl;
  imprimeLista1( p->enlace );
};

void imprimeLista2 (s_caja *p)
{
  if( p == NULL ) return;

  imprimeLista2( p->enlace );
  cout << p->dato << endl;
};

main() {
  s_caja *p = NULL, *q;
  int elem;
  cout << "Cuando quieras terminar, da un número negativo \n";
  do{
    cout << "dame un número entero: ";
    cin>>elem;
    if( elem >= 0 ){
      q = new s_caja;
      q->dato = elem;
      q->enlace = p;
      p = q;
    };
  }while( elem >= 0 );

  int opcion;
  // Recorrido de la lista
  do{
    cout << "Como deseas imprimir la pila? " << endl
         << "  0: en orden (LIFO) \n"
         << "  1: en orden inverso \n";
    cin >> opcion;
  }while( opcion<0 || opcion>1 );

  if( opcion == 0 ){
    cout << " Los elementos de la lista LIFO son: \n";
    imprimeLista1(p);
  }
  else{
    cout<<"Los elementos de la lista LIFO invertida son:\n";
    imprimeLista2(p);
  };

  // Desalojar la lista
```

```
  while( p != NULL ) {
    q = p->enlace;
    delete p;
    p = q;
  };

  return 0;
}
```

La forma de implementar un programa que solucione un problema determinado varía con la experiencia y personalidad del programador.

Ejemplo 8.3.- presentamos otro programa que hace lo mismo que la solución anterior, pero utilizando la función `Push()` para construir la pila. En este caso, el usuario indica con una "`s`" que sí desea agregar otro elemento, a diferencia del programa anterior, en el que el usuario indica con un número negativo, que ya no desea agregar otro elemento a la pila.

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

struct s_caja {
  int    dato;
  s_caja *enlace;
};

s_caja *p = NULL;

bool otra_caja() {
  char c;
  cout<< "\n Deseas otro elemento? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;
  q = new s_caja;
  if( q == NULL )
    return NULL;

  int elem;
  // pide los datos
  cout << "proporciona un numero entero: ";
  cin >> elem;
  q->dato = elem;
  q->enlace = NULL;

  return q;
}

void Push( s_caja *q ) {
  if( q == NULL )
    return;

  // "empuja" una caja
  q->enlace = p;
  p = q;
}

void imprimeLista1( s caja *q ) {
  if( q == NULL ){
    cout<< "NULL"<<endl;
    return;
  };

  cout << q->dato << endl;
  q = q->enlace;
  imprimeLista1(q );
};

void imprimeLista2( s_caja *q ) {
  if( q == NULL ){
    cout<< "NULL"<<endl;
```

```
    return;
  };

  imprimeLista2(q->enlace);

  cout << q->dato << endl;
};

int main() {
  s_caja *q;

  int elem;
  while( otra_caja())
    // crear y guardar
    Push( hacer_caja() );

  int opcion;
  // Recorrido de la lista
  do{
    cout << "Como deseas imprimir la pila? " << endl
         << "  0: en orden (LIFO) \n"
         << "  1: en orden inverso \n";
    cin >> opcion;
  }while( opcion<0 || opcion>1 );

  if( opcion == 0 ){
    cout << " Los elementos de la lista LIFO son: \n";
    imprimeLista1( p );
  }
  else{
    cout << " Los elementos de la LIFO inverstida son: \n";
    imprimeLista2( p );
  };

  // Desalojar la lista
  while( p != NULL ) {
    q = p->enlace;
    delete p;
    p = q;
  };

  return 0;
}
```

En la Figura VIII-6 se muestra un ejemplo de la salida de este programa imprimiendo la lista LIFO en orden.

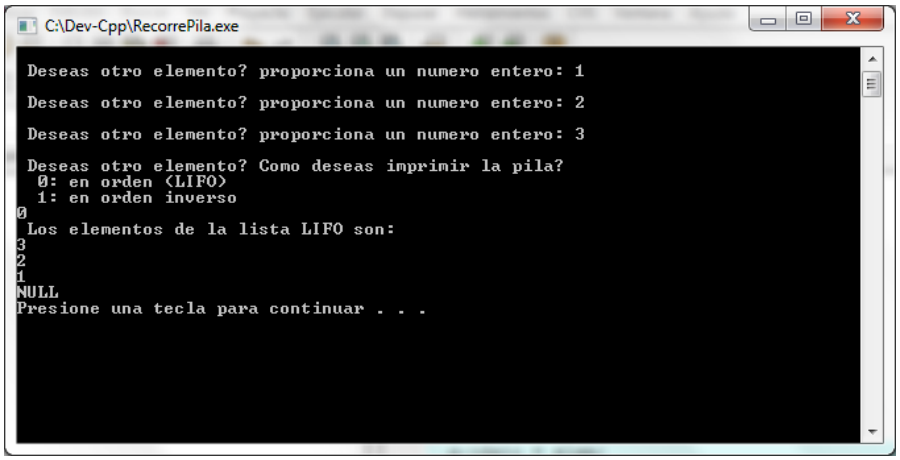


Figura VIII-6 Desplegado de una LIFO en orden.

En la Figura VIII-7 se muestra la salida del programa imprimiendo la LIFO invertida.

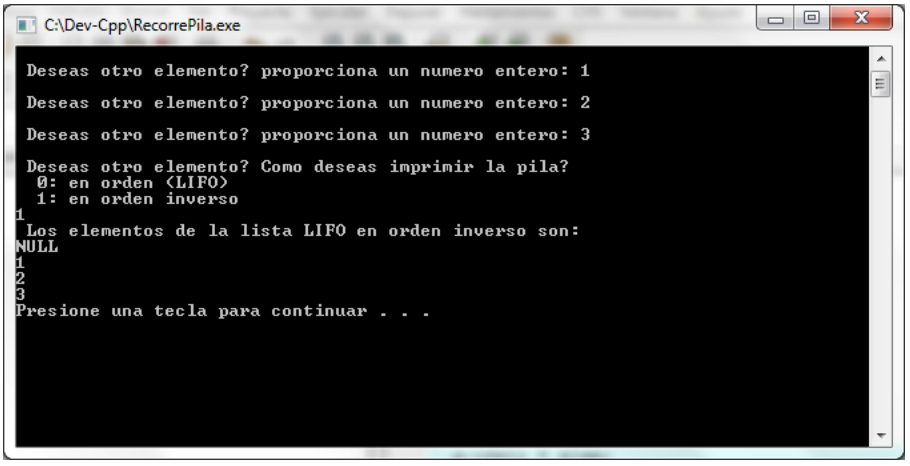


Figura VIII-7 Desplegado de una LIFO en orden inverso.

Ejercicio 8.4.- Modificar el programa anterior para el usuario construya una ***cola (queue)*** de elementos de tipo:

```
struct s_caja{
  int    dato;
  s_caja *siguiente;
};
```

y que pregunte al usuario si desea que se imprima el contenido de la *cola* en orden, o en orden inverso.

Solución:

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

struct s_caja {
  int    dato;
  s_caja *enlace;
};

s_caja *caja_inicial = NULL, *caja_final = NULL;

bool otra_caja() {
  char c;
  cout<< "\n Deseas otro elemento? ";
  c = getch();
  return c == 's' || c == 'S';
};

s_caja *hacer_caja() {
  // aloja memoria
  s_caja *q;
  q = new s_caja;
  if( q == NULL )
    return NULL;

  int elem;
  // pide los datos
  cout << "proporciona un numero entero: ";
  cin >> elem;
  q->dato = elem;
  q->enlace = NULL;

  return q;
}


void Formar( s_caja *q ) {
  if( q == NULL )
    return;
  q->enlace = NULL;
  // "forma" una caja al final
  if( caja_final == NULL )
    caja_inicial = q;
  else
    caja final->enlace = q;
  caja_final = q;
}


void imprimeLista1( s_caja *q ) {
  if( q == NULL ){
    cout<< "NULL"<<endl;
    return;
  };

  cout << q->dato << endl;
  q = q->enlace;
  imprimeLista1( q );
```

```
};

void imprimeLista2( s_caja *q ) {
  if( q == NULL ){
    cout<< "NULL"<<endl;
    return;
  };

  imprimeLista2( q->enlace );

  cout << q->dato << endl;
};

int main() {
  s_caja *q;

  int elem;
  while( otra_caja())
    // crear y guardar
    Formar( hacer_caja() );


  int opcion;
  // Recorrido de la lista
  do{
    cout << "Como deseas imprimir la cola? " << endl
         << "  0: en orden (FIFO) \n"
         << "  1: en orden inverso \n";
    cin >> opcion;
  }while( opcion<0 || opcion>1 );

  if( opcion == 0 ){
    cout << "La lista FIFO: \n";
    imprimeLista1( caja_inicial );
  }
  else{
    cout << " La lista FIFO en orden inverso: \n";
    imprimeLista2( caja_inicial );
  };

  // Desalojar la lista
  while( caja_inicial != NULL ) {
    q = caja_inicial->enlace;
    delete caja inicial;
    caja_inicial = q;
  };

  return 0;
}
```

VIII.3 Ordenamiento Recursivo

Algunos métodos de ordenamiento se basan en la recursión, es decir, aplican el mismo principio de ordenamiento a un sub-problema que a su vez se divide en partes más pequeñas que se ordenan de la misma manera que el problema original. Finalmente se llega a un caso base e inicia un proceso de *regreso* mediante el cual se logra que los elementos queden parcialmente ordenados, hasta llegar al problema original, en el que los elementos quedan totalmente ordenados.

VIII.3.1 Ordenamiento por mezcla (Merge Sort)

El algoritmo de ordenamiento por mezcla o fusión utiliza la técnica de Divide y Vencerás para realizar la ordenación de un vector. Su estrategia consiste en dividir un arreglo (también llamado vector) en dos sub arreglos (o sub vectores) que sean de un tamaño tan similar como sea posible, ordenar estas dos partes mediante llamadas recursivas, y finalmente combinar (mezclar) los dos partes ya ordenadas.

El algoritmo para mezclar es el siguiente: se parte el arreglo a la mitad, y se trabaja con dos índices: i_1 , i_2. Con i_1 se recorre la primera mitad del arreglo, con i_2 se van recorriendo los elementos de la segunda mitad del arreglo, como se ilustra en la Figura VIII-8. Se compara el contenido del arreglo en índice i_1 , con el contenido del índice i_2 , se elige el más pequeño (cuando se requiere orden ascendente) y se incrementa en uno el índice cuyo contenido se seleccionó.

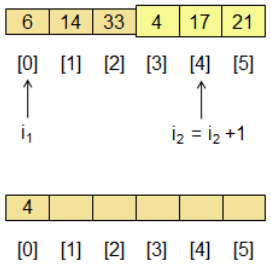


Figura VIII-8 ***La mezcla requiere dos índices, uno para la primera mitad y otro para la segunda mitad del arreglo.***

A continuación se presenta el código en C/C++ de la función de ordenamiento por mezcla. Recibe como parámetros el apuntador al arreglo y su tamaño. Se puede observar que es una función recursiva que se llama a sí misma en dos ocasiones, la primera ocasión envía la primera mitad del arreglo, cuando hace la segunda llamada se parte de la base de que la primera mitad ya quedó ordenada. La llamada a la función combinar se hace una vez que ya se tienen ordenadas tanto la primera mitad como la segunda.

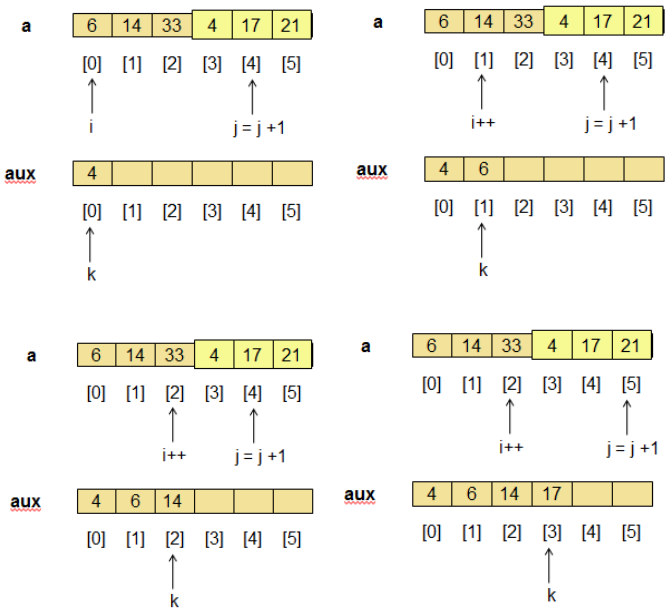
```
void merge_sort(int *a, int ini, int fin) {
  int med;
  if (ini < fin) {
    med = (ini + fin)/2;
    merge_sort(a, ini, med);
    merge_sort(a, med + 1, fin);
    mezcla(a, ini, med, fin);
  }
}
```

La función mezcla combina la primera mitad del arreglo (vector) con la segunda bajo el supuesto de que cada una de estas mitades ya está ordenada. Recibe como parámetros el apuntador al inicio del arreglo o vector, el índice al inicio del arreglo, el índice a la mitad del arreglo y el índice del fin del arreglo.

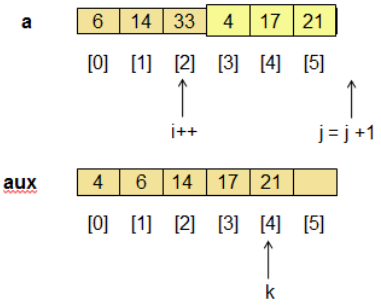
Se utiliza un arreglo auxiliar en el que se guarda la mezcla de la primera y la segunda mitad del arreglo recibido. Cuando se termina el procedimiento de mezcla, se copia el contenido del arreglo auxiliar al arreglo recibido.

```
void mezcla( int *a, int ini, int med, int fin ) {
  int *aux;
  aux = new int[fin - ini + 1];
  int i = ini;      // Índice sub vector izquierdo
  int j = med + 1; // Índice sub vector derecho
  int k = 0;       // Índice del vector aux

  // Mientras ninguno de los indices llegue a su fin se hacen
  // comparaciones. El elemento más pequeño se copia a "aux"
  while (i <= med && j <= fin) {
    if( a[i] < a[j] ) {
      aux[k] = a[i];
      i++;
    }
    else {
      aux[k] = a[j];
      j++;
    }
    k++;
  }

  // Uno de los dos sub-vectores ya ha sido copiado del todo,
  //  falta copiar el otro sub-vector
  while( i <= med ) {
    aux[k] = a[i];
    i++;
    k++;
  }

  while( j <= fin ) {
    aux[k] = a[j];
    j++;
    k++;
  }

  // Copiar los elementos ordenados de aux al vector "a"
  for( int m=0; m<fin-ini+1; m++ )
    a[ini + m] = aux[m];
  delete [] aux;
}
```

Ejemplo 8.4.- En la prueba de escritorio de la Figura VIII-9, se muestra la secuencia de la función mezcla para combinar las dos mitades (ya ordenadas) del arreglo de la Figura VIII-8.




```
void mezcla(int *a, int ini, int med, int fin) {
    int *aux;
    aux = new int[fin - ini + 1];
    int i = ini;       // Índice subvector izquierdo
    int j = med + 1; // Índice subvector derecho
    int k = 0;         // Índice del vector aux

    // Mientras ninguno de los indices llegue a su fin se hacen
    // comparaciones. El elemento más pequeño se copia a "aux"
    while (i <= med && j <= fin) {
        if (a[i] < a[j]) {
            aux[k] = a[i];
            i++;
        }
        else {
            aux[k] = a[j];
            j++;
        }
        k++;
    }
    .
    .
    .
```




```
void mezcla(int *a, int ini, int med, int fin) {
    ...
}
    // Uno de los dos sub-vectores ya ha sido copiado del todo,
    //  falta copiar el otro sub-vector
    while (i <= med) {
        aux[k] = a[i];
        i++;
        k++;
    }

    while (j <= fin) {
        aux[k] = a[j];
        j++;
        k++;
    }
```

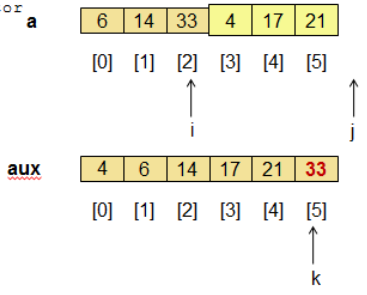


Figura VIII-9 *Mezcla de subvectores.*

En la prueba de escritorio de la función mezcla se observa como se elige el elemento menor de cada sub arreglo, y se incrementa el índice correspondiente, hasta que uno de los índices rebase el final del sub arreglo. Entonces se procede a copiar los demás elementos que falten del otro sub arreglo.

Ejemplo 8.5.- Hacer la prueba de escritorio para el algoritmo de *mergesort* con los siguientes datos de entrada.

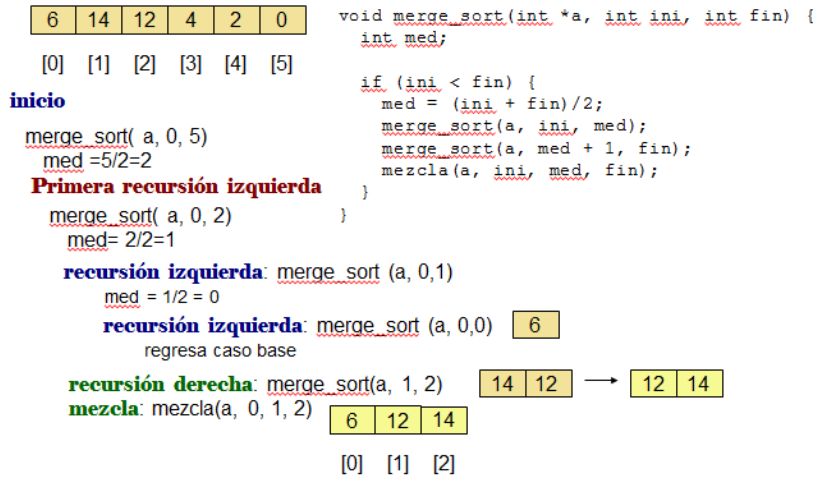


Figura VIII-10 Prueba de escritorio de la recursión del lado izquierdo

En la Figura VIII-10 se aprecia que, cuando el sub vector es de tamaño 1, se tiene el *caso base*. Al partir un vector de tamaño 3 a la mitad, queda un sub vector de tamaño 1 igual a {6} a la izquierda y un sub vector de tamaño 2 igual a {14, 12} a la derecha. Cuando se lleva a cabo el tercer nivel de recursión tenemos una vez más el caso base, así que en este segundo nivel se procede a combinar el "14" con el "12" y obtenemos el vector {12, 14}, al regresar al nivel anterior, se combina el {6} con el {12, 14} y se obtiene el sub vector de la mitad izquierda ordenado: {6, 12, 14}.en la Figura VIII-11 se muestra la mezcla entre los sub vectores {6} y {12,14}.

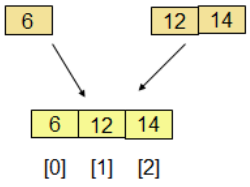


Figura VIII-11 Mezcla de los sub vectores {6} y {12,14}.

El procedimiento para ordenar el primer sub vector de la derecha es muy similar y se muestra en la prueba de escritorio de la Figura VIII-12.

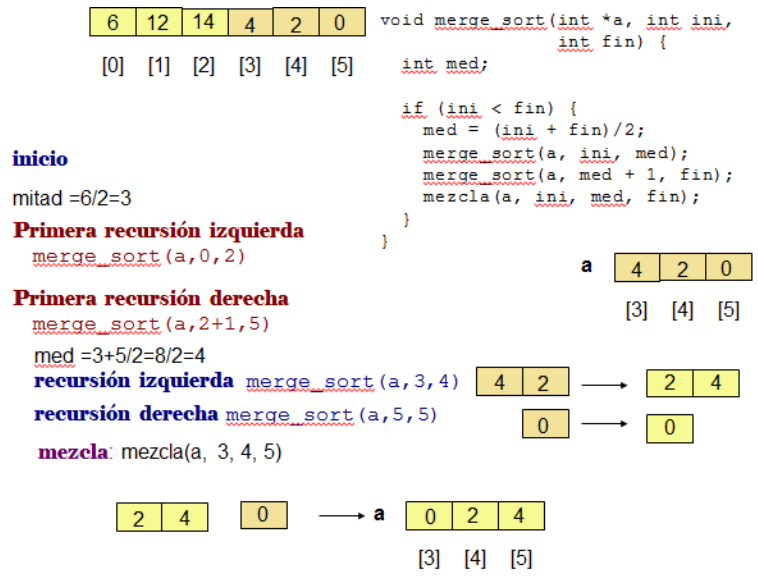


Figura VIII-12 *Prueba de escritorio de la recursión del lado derecho.*

Ahora ya tenemos los dos sub vectores principales ordenados por lo que solo resta mezclarlos como se indicó en la Figura VIII-11.

Para que el *ordenamiento por mezcla* sea eficiente, el tamaño de los dos sub vectores debe ser igual o muy parecido, ya que en el caso contrario, cuando un sub vector es más pequeño que el otro, el algoritmo se vuelve bastante ineficiente.

En el *ordenamiento por mezcla* el mayor esfuerzo se hace durante el proceso de regreso de la recursión, ya que es cuando se lleva a cabo la mezcla. Al dividir el problema no es necesario hacer muchos cálculos, ya que solo hay que partir los sub-arreglos por la mitad.

Ejercicio 8.5.- Hacer un programa que ordene por *Merge Sort* (ordenamiento por mezcla) un arreglo de enteros de tamaño variable.

Solución parcial:

```
#include <iostream>

void pedirDatosArreglo( int *A, int n ){
  for( int i=0; i<n; i++ ){
    cout << "A["<< i << "]=?";
    cin >> A[i];
  }
};

void mezcla( int *a, int ini, int med, int fin ) {
  int *aux;
  aux = new int[fin - ini + 1];
  int i = ini;      // Índice subvector izquierdo
  int j = med + 1; // Índice subvector derecho
  int k = 0;       // Índice del vector aux

  // Mientras ninguno de los indices llegue a su fin se hacen
  // comparaciones. El elemento más pequeño se copia a "aux"
  . . .

  // Uno de los dos sub-vectores ya ha sido copiado del todo,
  //  falta copiar el otro sub-vector
  while( i <= med ) {
    ...
  }

  while( j <= fin ) {
    ...
  }

  // Copiar los elementos ordenados de aux al vector "a"
  . . .

  delete [] aux;
}

void merge_sort( int *a, int ini, int fin ) {
  // Si ini = fin el sub-vector es de un solo elemento y
  // ya está ordenado por definición
  int med;
  if( ini < fin ) {
    // Considerar que el valor de med se redondea hacia abajo.
    med = (ini + fin)/2;

    merge_sort( a, . . ., . . . );
    merge_sort( a, . . ., . . . );
    mezcla( a, ini, . . ., . . .);
  }
}

int main()
{
  int opcion,n, *A;
  char c;

  cout << "Cuantos numeros (enteros) deseas ordenar?";
  cin>>n;
```

```
  // Alojar el arreglo de tamaño n;
  A = new int[n];

  pedirDatosArreglo(...);

  merge_sort(...);

  cout << "Los datos ordenados por Merge Sort: \n";
  for( int i=0; i<n; i++ )
     cout << "A["<< i << "]=" << A[i] << endl

  return 0;
}
```

Solución completa:

```
#include <iostream>

void pedirDatosArreglo( int *A, int n ){
  for( int i=0; i<n; i++ ){
    cout << "A["<< i << "]=?";
    cin >> A[i];
  }
};

void mezcla( int *a, int ini, int med, int fin ) {
  int *aux;
  aux = new int[fin - ini + 1];
  int i = ini;      // Índice subvector izquierdo
  int j = med + 1; // Índice subvector derecho
  int k = 0;       // Índice del vector aux

  // Mientras ninguno de los indices llegue a su fin se hacen
  // comparaciones. El elemento más pequeño se copia a "aux"
  while( i <= med && j <= fin ) {
    if( a[i] < a[j] ) {
      aux[k] = a[i];
      i++;
    }
    else {
      aux[k] = a[j];
      j++;
    }
    k++;
  }

  // Uno de los dos sub-vectores ya ha sido copiado del todo,
  //  falta copiar el otro sub-vector
  while( i <= med ) {
    aux[k] = a[i];
    i++;
    k++;
  }

  while( j <= fin ) {
    aux[k] = a[j];
    j++;
    k++;
  }

  // Copiar los elementos ordenados de aux al vector "a"
  for( int m=0; m<fin-ini+1; m++ )
    a[ini + m] = aux[m];
  delete [] aux;
}

void merge_sort( int *a, int ini, int fin ) {
  // Si ini = fin el sub-vector es de un solo elemento y
  // ya está ordenado por definición
  int med;
  if (ini < fin) {
  // Considerar que el valor de med se redondea hacia abajo.
    med = (ini + fin)/2;
```

```
    merge_sort(a, ini, med);
    merge_sort(a, med + 1, fin);
    mezcla(a, ini, med, fin);
  }
}

int main( int argc, char *argv[] )
{
  int opcion,n, *A;
  char c;

  cout << "Cuantos numeros (enteros) deseas ordenar?";
  cin>>n;
  // Alojar el arreglo de tamaño n;
  A = new int[n];

  pedirDatosArreglo(A,n);

  merge_sort(A,0,n-1);

  cout << "Los datos ordenados por Merge Sort: \n";
  for( int i=0; i<n; i++ )
    cout << "A["<< i << "]=" << A[i] << endl;

  return 0;
}
```

VIII.3.2 Ordenamiento rápido (Quick Sort)

El método de ordenamiento rápido, llamado quicksort está basado en la técnica de Divide y Vencerás, éste es un algoritmo que trabaja bien en casi todas las situaciones y consume en general menos recursos (memoria y tiempo) que otros métodos. A diferencia del mergesort el mayor esfuerzo se hace al construir los subcasos y no en combinar las soluciones. En el primer paso, el algoritmo selecciona como pivote uno de los elementos del arreglo a ordenar. El segundo paso es partir el arreglo por ambos lados del pivote: se desplazan los elementos de tal manera que los que sean mayores que el pivote queden a su derecha, y los demás quedan a su izquierda. Cada uno de los lados del pivote se ordena independientemente mediante llamadas recursivas al algoritmo, el resultado final es un arreglo completamente ordenado.

En la Figura VIII-13 se ilustra la forma en la que se hace la partición I_1 corresponde al pivote de la primera partición, I_2 son los respectivos pivotes de la parte izquierda y derecha de la segunda partición.

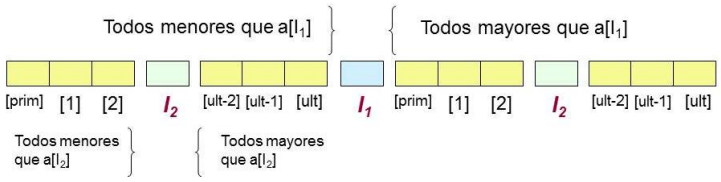


Figura VIII-13 El Quicksort.

A continuación el código en C/C++ de la función `quick_sort`. Recibe como parámetros el apuntador al arreglo a ordenar y los índices del inicio y del final del arreglo.

```
void quick_sort(int *a, int prim, int ult) {
  int pivote = particion(a, prim, ult);
  if (prim < pivote - 1)
     quick_sort(a, prim, pivote - 1);
  if (pivote < ult)
     quick_sort(a, pivote, ult);
}
```

La función `particion(a,prim,ult)` regresa el índice del elemento pivote, de tal forma que todos los elementos de la izquierda son menores que `a[pivote]` y todos los elementos de la derecha son mayores que `a[pivote]`.

Los parámetros que recibe la función `particion` son los siguientes:

```
particion(a, prim, ult)

int particion( int *a, int prim, int ult )
```

Apuntador al inicio del sub-arreglo

Índice del primer elemento del sub-arreglo

Índice del último elemento del sub-arreglo

Ejemplo 8.6.- A continuación, en la Figura VIII-14 se ilustra cómo funciona `particion` para el caso de la primera llamada recursiva.

```
int particion(int *a, int izq, int der){
  int i = izq, j = der;
  int tmp;
  int pivote = a[(izq + der) / 2];

  while (i <= j) {
     while (a[i] < pivote)
        i++;
     while (a[j] > pivote)
        j--;
     if (i <= j) {
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
        i++;
        j--;
     } ;
  };

  return i;
}
```

pivote = 4 a[pivote] = 12

pivote = a[(0+5)/2]=a[2]=12

i = 0 j = 5

6	14	12	4	2	0
[0]	[1]	[2]	[3]	[4]	[5]

i =0,1,... mientras que a[i] < pivote i = 1

j =5,4,... mientras que a[j] > pivote j = 5

i = 2
j = 4

6	0	12	4	2	14
[0]	[1]	[2]	[3]	[4]	[5]

i =2,3,... mientras que a[i] < pivote i = 2

j =4,... mientras que a[j] > pivote j = 4

i = 3
j = 3

6	0	2	4	12	14
[0]	[1]	[2]	[3]	[4]	[5]

i = 3... mientras que a[i] < pivote i = 4

j = 3,... mientras que a[j] > pivote j = 3

i = 4
j = 3

6	0	2	4	12	14
[0]	[1]	[2]	[3]	[4]	[5]

Figura VIII-14 La función `particion`.

En el *ordenamiento rápido* el mayor esfuerzo se hace en la partición. Durante el proceso de regreso de la recursión no es necesario hacer muchos cálculos.

Ejercicio 8.6.- Completa el siguiente programa el cual permite que desde el programa principal el usuario pueda elegir el método de ordenamiento.

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

void pedirDatosArreglo( int *A, int n ){
  for( int i=0; i<n; i++ ){
    cout << "A["<< i << "]=?";
    cin >> A[i];
  };
}

int particion( int *a, int izq, int der ){
  int i = izq, j = der;
  int tmp;
  int pivote = a[(izq + der) / 2];
  ...

  return i;
}


void quick_sort( int *a, int prim, int ult ) {
  int pivote = particion(...);

  if( prim < pivote - 1 )
    quick_sort(...);
  if( pivote < ult )
    quick_sort(...);
}

void mezcla( ... ) {
  int *aux;
  ...
}

void merge_sort( ... ) {
  ...
}

int main()
{
  int opcion,n, *A;
  char c;

  do
  {
    cout << "Cuantos numeros (enteros) deseas ordenar?";
    cin>>n;
    // Alojar el arreglo de tamaño n;
    ...;

    pedirDatosArreglo(...);

    do {
      cout << endl
           << "¿Opción para ordenarlos (menor a mayor)?: \n"
           << " 1 : QuickSort " <<endl
           << " 2 : MergeSort " <<endl;
```

```
      cin >> opcion;
    }while( ( opcion < 1 )||( opcion > 2 ));

    switch(opcion)
    {
      case 1: ...;
              cout << "\n Los datos con Quick Sort son: \n";
              break;

      case 2: ...;
              cout << "\n Los datos con Merge Sort son: \n";
              break;
    };

    for( int i=0; i<n; i++ )
      cout << "A["<< i << "]=" << A[i] << endl;

    cout<< "oprime ""Esc"" para terminar" <<endl;
    c = getch();

  }while(27 != c);
  cout << " Adios! ";

  return 0;
}
```

Solución:

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

void pedirDatosArreglo( int *A, int n ){
  for( int i=0; i<n; i++ ){
    cout << "A["<< i << "]=?";
    cin >> A[i];
  };
}

int particion( int *a, int izq, int der ){
  int i = izq, j = der;
  int tmp;
  int pivote = a[(izq + der) / 2];

  while( i <= j ) {
    while( a[i] < pivote )
      i++;
    while( a[j] > pivote )
      j--;
    if( i <= j ) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++;
      j--;
    } ;
  };

  return i;
}


void quick_sort( int *a, int prim, int ult ) {
  int pivote = particion(a, prim, ult);

  if( prim < pivote - 1 )
    quick_sort( a, prim, pivote - 1 );
  if( pivote < ult )
    quick_sort( a, pivote, ult );
}


void mezcla( int *a, int ini, int med, int fin ) {
  int *aux;
  aux = new int[fin - ini + 1];
  int i = ini;      // Índice subvector izquierdo
  int j = med + 1; // Índice subvector derecho
  int k = 0;       // Índice del vector aux

  // Mientras ninguno de los indices llegue a su fin se hacen
  // comparaciones. El elemento más pequeño se copia a "aux"
  while( i <= med && j <= fin ) {
    if( a[i] < a[j] ) {
      aux[k] = a[i];
      i++;
    }
```

```
    else {
      aux[k] = a[j];
      j++;
    }
    k++;
  }

  // Uno de los dos sub-vectores ya ha sido copiado del todo,
  //  falta copiar el otro sub-vector
  while( i <= med ) {
    aux[k] = a[i];
    i++;
    k++;
  }

  while( j <= fin ) {
    aux[k] = a[j];
    j++;
    k++;
  }

  // Copiar los elementos ordenados de aux al vector "a"
  for( int m=0; m<fin-ini+1; m++ )
    a[ini + m] = aux[m];
  delete [] aux;
}

void merge_sort( int *a, int ini, int fin ) {
  // Si ini = fin el sub-vector es de un solo elemento y,
  // por lo tanto ya está ordenado por definición
  int med;
  if (ini < fin) {
  // Considerar que el valor de med se redondea hacia abajo.
    med = (ini + fin)/2;

    merge_sort(a, ini, med);
    merge_sort(a, med + 1, fin);
    mezcla(a, ini, med, fin);
  }
}

int main()
{
  int opcion,n, *A;
  char c;

  do {
    cout << "Cuantos numeros (enteros) deseas ordenar?"; cin>>n;
    // Alojar el arreglo de tamaño n;
    A = new int[n];

    pedirDatosArreglo(A,n);

    do {
      cout << endl
           << "¿Opción para ordenarlos (menor a mayor)?: \n"
           << " 1 : QuickSort " <<endl
```

```
            << " 2 : MergeSort"    <<endl;
      cin >> opcion;
    }while( ( opcion < 1 )||( opcion > 2 ) );

    switch(opcion)
    {
      case 1: quick_sort(A, 0, n-1);
              cout << "\n Los datos con Quick Sort son: \n";
              break;

      case 2: merge_sort(A, 0, n-1);
              cout << "\n Los datos con Merge Sort son: \n";
              break;
    };

    for( int i=0; i<n; i++ )
      cout << "A["<< i << "]=" << A[i] << endl;

    cout<< "oprime ""Esc"" para terminar" <<endl;
    c = getch();

  }while(27 != c);
  cout << " Adios! ";

  return 0;
}
```

VIII.4 Árboles

VIII.4.1 Definición de árbol.

Un grafo es un conjunto de objetos llamados *vértices* o *nodos* unidos por enlaces llamados *aristas* o *arcos*, que permiten representar relaciones binarias entre elementos de un conjunto. Un árbol es un grafo con un nodo raíz, aristas (o ramás) que conectan un nodo con otro estableciendo una relación de padre-hijo con la condición de que cada hijo tenga un solo padre. Un ejemplo de árbol se ilustra en la Figura VIII-15.

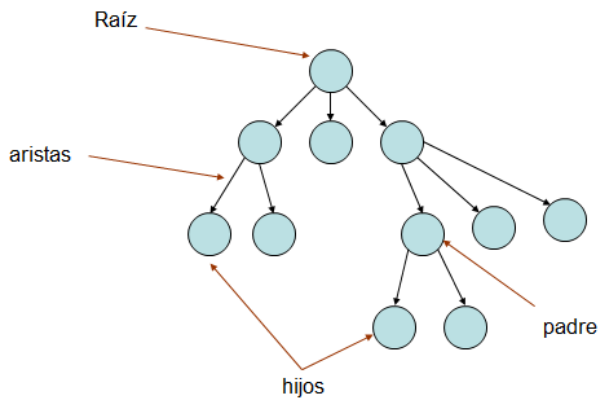


Figura VIII-15 Partes de un árbol.

Además un árbol tiene las siguientes características:

- No existen nodos aislados (es un grafo conectado).
- Cada nodo tiene 0 o más nodos hijos hacia los cuales tiene una arista dirigida.
- Hay un solo nodo que no tiene padre al cual se le llama *nodo raíz*.
- Cada nodo tiene exactamente un padre (excepto el *nodo raíz*).
- A los nodos que no tienen hijos se les denomina *hojas*.
- Los nodos hijos de un mismo padre son *hermanos*.
- Los nodos que no son hojas son *nodos internos* o *nodos no terminales*.
- No hay ciclos.

La construcción de árboles es uno de las aplicaciones en las que la recursión simplifica el problema. Como se ilustra en la Figura VIII-16, un árbol se puede describir recursivamente de la siguiente manera:

- El inicio del *árbol* es un nodo al que se le llama *nodo raíz*.
- El *nodo raíz* tiene 0 o más nodos hijos tales que cada uno de estos es el *nodo raíz* de otro *árbol*.
-

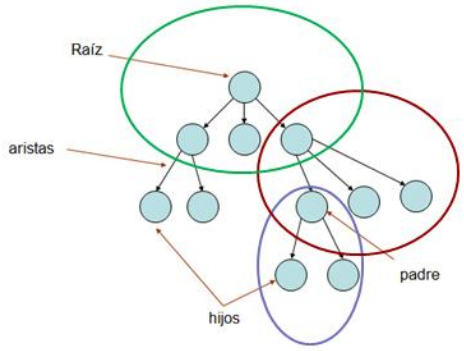


Figura VIII-16 *Definición recursiva de un árbol. Cada nodo es a su vez un árbol también llamado subárbol.*

Cuando se trabaja con árboles se utilizan los siguientes términos (los cuales se ilustran en la Figura VIII-17).

- *Camino* es una secuencia de ramás contiguas que van de un nodo n_x a otro nodo n_y.
- *Longitud de un camino* es el número de ramás entre el nodo n_x y el nodo n_y.
- *Nivel* de un nodo es la longitud del camino que lo conecta con el nodo raíz.
- *Profundidad del árbol* es la longitud del camino más largo que conecta el nodo raíz con una hoja.
-

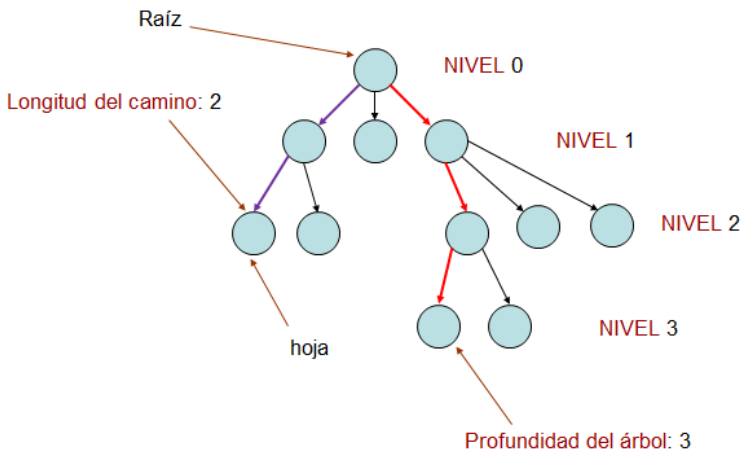


Figura VIII-17 *Términos para definir a un árbol.*

Un árbol puede verse como una estructura de datos en la que cada nodo contiene datos y enlaces hacia cada nodo hijo. De esta manera el árbol se crea y se maneja de forma similar a una lista ligada. Las estructuras en forma de árboles sirven para almacenar grandes cantidades de datos, así tanto el almacenamiento como la búsqueda de los datos es más eficiente que si se almacena de forma

secuencial. Otras de las aplicaciones de las estructuras en forma de árboles son el diseño de compiladores y el procesamiento de textos.

VIII.4.2 Árboles binarios.

Un ***árbol binario*** es aquel en el que los nodos solo pueden tener 0, 1 o 2 hijos. Se dinstingue entre subárbol izquierdo y derecho a cada uno de los hijos, si existen. Véase la Figura VIII-18.

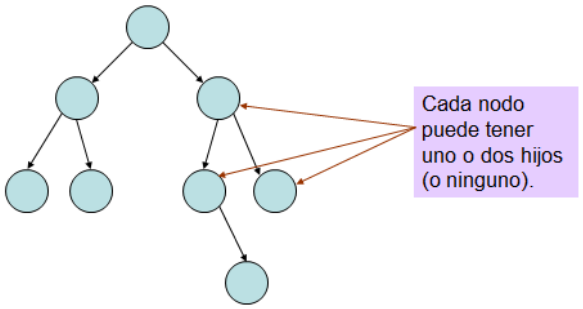


Figura VIII-18 Estructura general de un Árbol Binario.

VIII.4.3 Construcción recursiva de un árbol.

Para construir un árbol binario se utiliza el siguiente procedimiento recursivo.

Se construye primero el nodo raíz.
Se construyen sus hijos.
Se asigna un apuntador desde el padre hacia cada hijo.

Para construir los hijos de los hijos se usa el mismo procedimiento.

En la siguiente función `ConstruyeArbol` los nodos del árbol tienen la estructura:

```
struct s_caja
{
  int elemento;
  s_caja *hijo_izq;
  s_caja *hijo_der;
}
```

La construcción recursiva del árbol se basa en un principio muy sencillo: hay que alojar un nodo raíz, en caso de que haya elementos del lado izquierdo, se construye el sub-árbol izquierdo siguiendo el mismo procedimiento, posteriormente, si hay elementos del lado derecho, entonces se construye el subárbol derecho aplicando el mismo principio. El proceso llega al caso base en cada una de las hojas del árbol. Después se inician los procesos de regreso entregando el apuntador al nodo raíz (para las hojas, son raíces sin subárboles).

```
s_caja * ConstruyeArbol()
{
  s_caja * raiz; // apuntador al nodo raíz
  raiz = new s caja;
  cout << "dato?";
  cin >> raiz->elemento;
  raiz->hijo_izq = NULL;
  raiz->hijo_der = NULL;
  if( hay_hijo(0) )
    raiz->hijo_izq = ConstruyeArbol(); //hacer sub-árbol izq.
  if( hay_hijo(1) )
    raiz->hijo_der = ConstruyeArbol(); //hacer sub-árbol der.
  return raiz;
}
```

La función `ConstruyeArbol` se apoya en la función:

```
bool hay_hijo( int n )
{
  char *lado[2] = {"izq","der"};
  cout << "hay hijo "
       << lado[n] <<"?";
  char c = getch();
  return c == 's' || c == 'S';
}
```

La cual es verdadera cuando el usuario indica que si hay un hijo en el nodo. El "0" es para preguntar por un hijo del lado izquierdo y el "1" para preguntar por un hijo del lado derecho.

VIII.4.4 Recorridos en un árbol.

El *recorrido* de un árbol binario consiste en visitar todos sus nodos. Existen tres maneras de hacer un recorrido en un árbol binario: *preorden*, *postorden* y *entreorden*.

Preorden.- En este recorrido, se visita primero el nodo raíz, a continuación se visita el subárbol izquierdo y posteriormente el subárbol derecho.

Por ejemplo, el recorrido en *preorden* del árbol binario de la *figura 8.20* es el siguiente: 29, 18, 99, 1, 0, 33, 59, 77.

Postorden.- En este recorrido se comienza por la hoja izquierda del subárbol izquierdo y el regreso se hace pasando primero por el lado derecho. La raíz se visita al final.

Por ejemplo, el recorrido en *postorden* del árbol binario de la *figura 8.20* es el siguiente: 99, 1, 18, 59, 33, 77, 0, 29.

Entreorden.- Se visita primero el subárbol izquierdo, después la raíz y al final el subárbol derecho.

Por ejemplo, el recorrido en *entreorden* del árbol binario de la Figura VIII-19 es el siguiente: 99, 18, 1, 29, 33, 59, 0, 77.

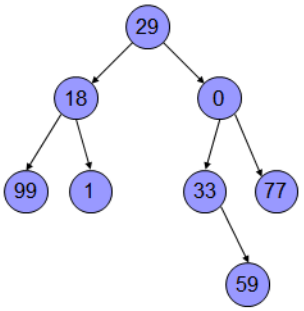


Figura VIII-19 Árbol binariocon datos en sus nodos.

Ejemplo 8.7.- Cuales son los recorridos en preorden, postorden y entreorden del árbol binario de la Figura VIII-20:

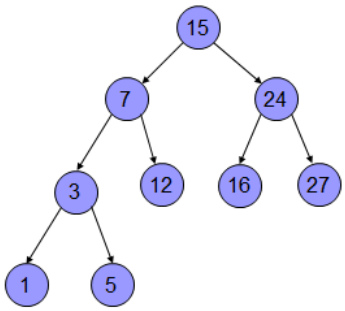


Figura VIII-20 Árbol binario para el ejemplo 1.

Solución:

- Recorrido en preorden (raíz-izquierda-derecha): 15, 7, 3, 1, 5, 12, 24, 16, 27.

- Recorrido en postorden (izquierda-derecha-raíz): 1, 5, 3, 12, 7, 16, 27, 24, 15.

- Recorrido en entreorden (izquierda-raíz-derecha): 1, 3, 5, 7, 12, 15, 16, 24, 27.

VIII.4.4.1 Implementación de recorridos en un árbol binario

Recorrido en preorden.

La siguiente función recursiva imprime los nodos de un árbol binario haciendo un recorrido en *preorden* y recibe como parámetro un apuntador al nodo raíz. Primero se imprime el nodo raíz, en la primera llamada recursiva se envía como parámetro el apuntador al hijo izquierdo del nodo, que a su vez se convierte en la raíz del siguiente subárbol, el caso base de la recursión se alcanza cuando se llega a la hoja de la extrema izquierda del árbol. En el regreso de la recursión se van imprimiendo los nodos de la derecha.

```
void imprimeArbol_preOrden( s_caja * p )
{
  if( p == NULL ) return;

  cout << " "<< p->elemento << ", ";
  imprimeArbol_preOrden( p->hijo_izq );
  imprimeArbol_preOrden( p->hijo_der );
}
```

Recorrido en postorden.

La siguiente función recursiva imprime los nodos de un árbol binario haciendo un recorrido en *postorden* y recibe como parámetro un apuntador al nodo raíz. En la primera llamada recursiva se envía como parámetro el apuntador al hijo izquierdo del nodo, que a su vez se convierte en la raíz del siguiente subárbol, el caso base de la recursión se alcanza cuando se llega a la hoja de la extrema izquierda del árbol, se imprime el nodo de la izquierda y en el regreso de la recursión se imprime el nodo de la derecha y después el nodo raíz.

```
void imprimeArbol_postOrden( s_caja * p )
{
  if( p == NULL ) return;

  imprimeArbol_postOrden( p->hijo_izq );
  imprimeArbol_postOrden( p->hijo_der );
  cout << " "<< p->elemento << ", ";
}
```

Recorrido en entreorden.

La siguiente función recursiva imprime los nodos de un árbol binario haciendo un recorrido en *entreorden* y recibe como parámetro un apuntador al nodo raíz. En la primera llamada recursiva se envía como parámetro el apuntador al hijo izquierdo del nodo, que a su vez se convierte en la raíz del siguiente subárbol, el caso base de la recursión se alcanza cuando se llega a la hoja de la extrema izquierda del árbol, se imprime el nodo de la izquierda y en el regreso de la recursión se imprime el nodo raíz y al final el nodo de la derecha.

```
void imprimeArbol_entreOrden( s_caja * p )
{
  if( p == NULL ) return;

  imprimeArbol_entreOrden( p->hijo_izq );
  cout << " "<< p->elemento << ", ";
  imprimeArbol_entreOrden( p->hijo_der );
}
```

VIII.4.5 Árboles binarios de búsqueda.

Un árbol binario de búsqueda es aquel en el que, dado un nodo cualquiera del árbol, todos los datos almacenados en el subárbol izquierdo son menores que el dato almacenado en este nodo, mientras que todos los datos almacenados en el

subárbol derecho son mayores que el dato almacenado en este nodo. En la Figura VIII-21 se ilustra un árbol binario de búsqueda y uno que no lo es.

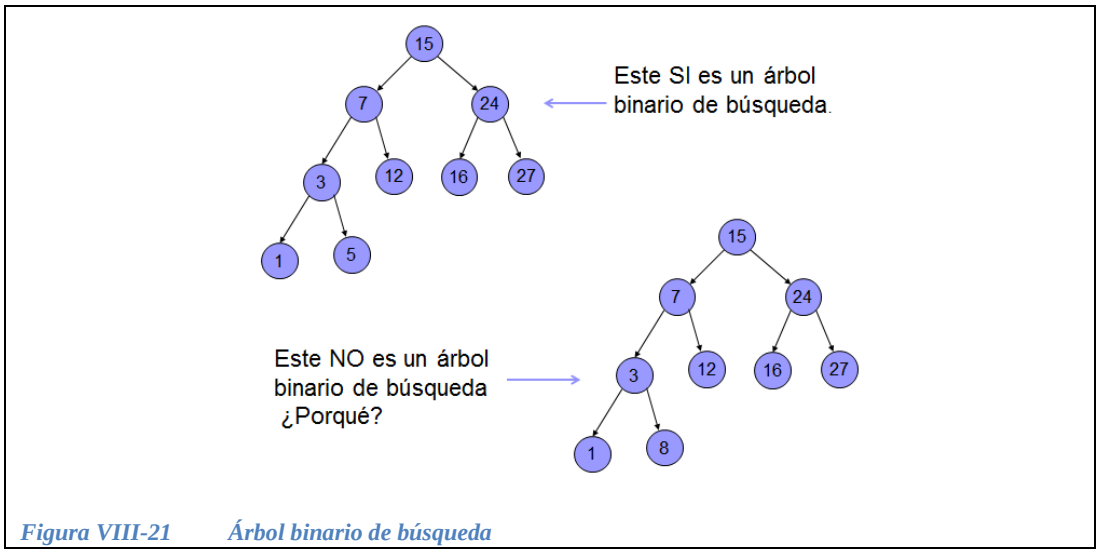


Figura VIII-21 Árbol binario de búsqueda

El árbol binario que no es de búsqueda contiene un nodo con el elemento 8 en la rama izquierda del nodo con el número 3. El nodo 3 junto con su ramás constituyen el subárbol derecho del nodo con el elemento 7, como 8 > 7 el árbol no cumple con la definición de árbol binario de búsqueda.

VIII.4.6 Ejercicios con árboles.

Ejercicio 8.7.- Hacer un programa que construya y capture los datos de un árbol binario en pre-orden, es decir primero el dato en el nodo y luego en los subárboles izquierdo y derecho. En cada nodo se le pregunta al usuario si ese nodo tiene hijo izquierdo e hijo derecho. Codificar las funciones para escribir los datos del árbol en pantalla en pre-orden, post-orden y entre-orden.

Solución parcial:

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

struct s_caja {
  int elemento;
  s_caja ...;
  s_caja ...;

};


bool hay_hijo( int n ) {
  char *lado[2] = {"izq","der"};
  cout << " hay hijo "
       << lado[n] << "?";
  char c = getch();

  return( c == 's' || c == 'S' );
}

s_caja * ConstruyeArbol() {
  s_caja * raiz;
  raiz = new s_caja;
  cout << " dato? ";
  cin >> raiz->elemento;
  raiz->hijo_izq = NULL;
  raiz->hijo_der = NULL;

  if( hay_hijo(...) )
    raiz->hijo_izq = ConstruyeArbol();
  if( hay_hijo(...) )
    raiz->hijo_der = ConstruyeArbol();

  return raiz;
}

void DestruyeArbol( s_caja *p ) {
  if( p == NULL ) return;
  DestruyeArbol( ... );
  DestruyeArbol( ... );
  delete p;
}

void imprimeArbol preOrden( s caja * p ) {
  if( p == NULL ) return;

  cout << " "<< p->elemento << ", ";
  imprimeArbol_preOrden( p->hijo_izq );
  imprimeArbol_preOrden( p->hijo_der );
}

void imprimeArbol_postOrden( s_caja * p ) {
  if( p == NULL ) return;

  . . .
}
```

```
void imprimeArbol_entreOrden( s_caja * p ) {
  if( p == NULL ) return;

  . . .
}

main() {
  s_caja *inicio;
  int opcion;

  cout<<"Ingresa los nodos del arbol en pre-orden \n";
      <<"Para cada nodo indicar si tiene un subarbol izq. y der"
      << " ""S"" para indicar que si, cualquier otra tecla para"
      << " indicar que no \n";

     . . . = ConstruyeArbol();

  while(true){
    do{
      cout << " Para imprimir el arbol, indica el recorrido: \n"
           << "  1: pre-orden: raíz-izq-der \n"
           << "  2: post-orden: izq-der-raíz \n"
           << "  3: entre-orden: izq-raíz-der \n"
           << "  4: salir \n";
       cin >> opcion;
    }while( opcion < 1 || opcion > 4 );

    switch( opcion ){
      case 1: cout << " El arbol recorrido en pre-orden es: \n";
              imprimeArbol_preOrden(. . .);
              break;
      case 2: cout <<" El árbol recorrido en post-orden es: \n";
              . . .
              break;
      case 3: cout<<" El arbol recorrido en entre-orden es:\n";
                . . .
              break;
      case 4: DestruyeArbol(. . .);
              return 0;
    };
  };
}
```

Solución completa:

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

using namespace std;

struct s_caja {
  int elemento;
  s_caja *hijo_izq;
  s_caja *hijo_der;

};

bool hay_hijo( int n ) {
  char *lado[2] = {"izq","der"};
  cout << " hay hijo "
       << lado[n] << "?";
  char c = getch();

  return(c == 's' || c == 'S');

}

s_caja * ConstruyeArbol() {
  s_caja * raiz;
  raiz = new s_caja;
  cout << " dato? ";
  cin >> raiz->elemento;
  raiz->hijo_izq = NULL;
  raiz->hijo_der = NULL;

  if( hay_hijo(0) )
    raiz->hijo_izq = ConstruyeArbol();
  if( hay_hijo(1) )
    raiz->hijo_der = ConstruyeArbol();

  return raiz;
}

void DestruyeArbol( s_caja *p ) {
  if( p == NULL ) return;
  DestruyeArbol( p->hijo_izq );
  DestruyeArbol( p->hijo_der );
  delete p;
}

void imprimeArbol_preOrden( s_caja * p ) {
  if( p == NULL ) return;

  cout << " "<< p->elemento << ", ";
  imprimeArbol_preOrden( p->hijo_izq );
  imprimeArbol_preOrden( p->hijo_der );
}

void imprimeArbol_postOrden( s_caja * p ) {
  if( p == NULL ) return;

  imprimeArbol_postOrden( p->hijo_izq );
```

```
  imprimeArbol_postOrden( p->hijo_der );
  cout << " "<< p->elemento << ", ";
}

void imprimeArbol_entreOrden( s_caja * p ) {
  if( p == NULL ) return;

  imprimeArbol_entreOrden( p->hijo_izq );
  cout << " "<< p->elemento << ", ";
  imprimeArbol_entreOrden( p->hijo_der );
}

main() {
  s_caja *inicio;
  int opcion;

  cout<<"Ingresa los nodos del arbol en pre-orden \n";
      <<"Para cada nodo indicar si tiene un subarbol izq. y der"
      << " ""S"" para indicar que si, cualquier otra tecla para"
      << " indicar que no \n";

  inicio = ConstruyeArbol();

  while(true){
    do{
      cout << " Para imprimir el arbol, indica el recorrido: \n"
           << "  1: pre-orden: raíz-izq-der \n"
           << "  2: post-orden: izq-der-raíz \n"
           << "  3: entre-orden: izq-raíz-der \n"
           << "  4: salir \n";
       cin >> opcion;
    }while( opcion < 1 || opcion > 4);

    switch(opcion){
      case 1: cout << " El arbol en pre-orden es: \n";
              imprimeArbol_preOrden(inicio);
              break;
      case 2: cout << " El árbol en post-orden es: \n";
              imprimeArbol_postOrden(inicio);
              break;
      case 3: cout << " El arbol en entre-orden es: \n";
              imprimeArbol_entreOrden(inicio);
              break;
      case 4: DestruyeArbol( inicio );
              return 0;
    };
  };
}
```

Ejercicio 8.8.- Agregar al programa anterior una función que reciba como parámetro un apuntador al nodo raíz de un árbol binario y también un número entero. Hacer una función que recorra el árbol en preorden en busca de un número y diga si está o no.

Solución:

```
bool BuscaEnArbol( s_caja *p, num ) {
  if( p == NULL )
    return false;

  if( p->elemento == num )
    return true;

  if( BuscaEnArbol( p->hijo_izq ))
    return true;

  if( BuscaEnArbol( p->hijo_der ))
    return true;

  return false;
}
```

Ejercicio 8.9.- Suponiendo que se tiene un árbol binario en el que el orden de los datos guardados coincide con el recorrido en entreorden, hacer una función que busque un número en el árbol mediante búsqueda binaria y, si lo encuentra, que regrese un apuntador al nodo donde está ese número, de lo contrario deberá regresar un apuntador nulo.

Solución:

```
s_caja * BusquedaBinaria( s_caja *p, num ) {
  if( p == NULL )
    return NULL;

  if( num == p->elemento)
    return p;

  if( num < p->elemento )
    return BusquedaBinaria( p->hijo_izq, num );

  if( num > p->elemento )
    return BusquedaBinaria( p->hijo_der, num );

}
```

Ejercicio 8.10.- Hacer un programa que capture los datos de un árbol binario en pre-orden y que la función imprime muestre en pantalla los datos del árbol en preorden junto con un número que indique en nivel de profundidad que cada dato tiene dentro del árbol.

Sugerencia: usar un parámetro adicional en la función imprime que indique el nivel.

Solución parcial:

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

struct s_caja {
  int elemento;
  s_caja *hijo_izq;
  s_caja *hijo_der;
};

bool hay_hijo( int . . . ) {
  char *lado[2] = {"izq","der"};
  cout << "hay hijo "
       << lado[n] <<"?\n";
  char c = getch();
  return c == 's' || c == 'S';
};

s_caja *captura() {
  s_caja * raiz;
  raiz = new s_caja;
  cout << "elemento? \n";
  cin >> raiz->elemento;
  raiz->hijo_izq = NULL;
  raiz->hijo_der = NULL;

  if( hay_hijo(0) )
    raiz->hijo_izq = captura();
  if( hay_hijo(1) )
    raiz->hijo_der = . . .;
  return raiz;
};

void imprime( s_caja *r, int nivel ) {
  if ( r == NULL ) return;
  cout << "nivel: " << nivel << "  elemento: "
       << r->elemento << endl;
  nivel++;
  imprime( r->hijo_izq, . . . );
  imprime( r->hijo_der, . . . );
};

main() {
  s_caja *raiz;
  cout << "proporciona los datos en preorden \n";
  . . . = captura();
  imprime ( . . ., . . .);

  return 0;
}
```

Solución:

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

struct s_caja {
  int elemento;
  s_caja *hijo_izq;
  s_caja *hijo_der;
};

bool hay_hijo( int n ) {
  char *lado[2] = {"izq","der"};
  cout << "hay hijo "
       << lado[n] <<"?\n";
  char c = getch();
  return c == 's' || c == 'S';
};

s_caja *captura ( ) {
  s_caja * raiz;
  raiz = new s_caja;
  cout << "elemento? \n";
  cin >> raiz->elemento;
  raiz->hijo_izq = NULL;
  raiz->hijo_der = NULL;

  if( hay_hijo(0) )
    raiz->hijo_izq = captura();
  if( hay_hijo(1) )
    raiz->hijo_der = captura();
  return raiz;
};

void imprime( s_caja *r, int nivel ) {
  if ( r == NULL ) return;
  cout << "nivel: " << nivel << "  elemento: "
       << r->elemento << endl;
  nivel++;
  imprime( r->hijo_izq, nivel );
  imprime( r->hijo_der, nivel );
};

main() {
  s_caja *raiz;
  cout << "proporciona los datos en preorden \n";
  raiz = captura();
  imprime( raiz, 0);

  return 0;
}
```

Un ejemplo de salida de este programa cuando el árbol se construye con los datos de uno de los árboles de la Figura VIII-21es el de la Figura VIII-22.

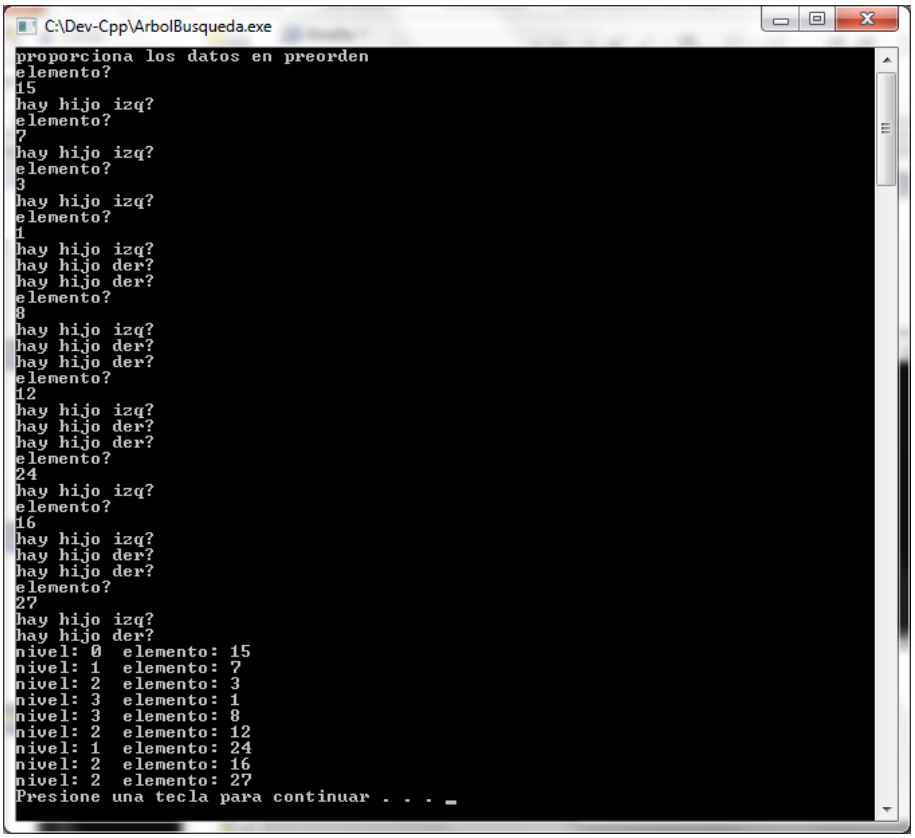


Figura VIII-22 Salida del programa del ejercicio 8.4.3.3.

Bibliografía

1. Aho, A. V. Hopcroft, J. E. y Ullman, J. D. “Estructura de Datos y Algoritmos”. Addison-Wesley Iberoamericana, México, 1998.

2. Antonakos, J. L. y Mansfield, K. C., “Programación estructurada en C”, Prentice Hall, México, 2000.

3. Bandura A., “Social foundations of thought and action”. Ed. Prentice Hall, 1986.

4. Cairó O. “Metodología de la programación, algoritmos, diagramas de flujo y programas”. Alfaomega, 2005.

5. Ceballos F. J., “Curso de programación C/C++”, Ed. RA-MA, 1995.

6. Deitel y Deitel, “C++ Como programar”, Ed. Pearson Prentice may, 2ª ed., 1999.

7. Denti, E., “Corso di Fondamenti di Informatica LA.” Facolta’ di Ingegneria, Universita’ degli Studi di Bologna, 2000.

8. Eckel, “Aplique C++”, Mc Graw-Hill / Interamericana de España, 1991.

9. Eggen P. y Kauchak D., “Estrategias docentes. Enseñanza de contenidos curriculares y desarrollo de habilidades de pensamiento”. Fondo de cultura económica, México, 1999.

10. Jamsa K., “Aprenda y practique C++”. 3era edición. Ed. Oxford 1999.

11. Joyanes Aguilar L., Sánchez García L., Zahonero Martínez I. “Estructura de Datos en C++”. Universidad Pontificia de Salamanca. 2007.

12. Joyanes Aguilar L., Sánchez L., “Programación en C++: un enfoque práctico”. McGraw-Hill, España, 2006.

13. Joyanes Aguilar L., “Fundamentos de programación", McGraw Hill/ Interamericana de España, 2008.

14. Kernighan B. y Ritchie D., “El lenguaje de programación C”, 2ª ed., Pearson Educación, México, 1991.

15. Knuth, D. E. “The Art of Computer Programming”, Vol-1 y 3; Addison Wesley Professional, New York, 1998.

16. Langsam, Augenstein y Tenenbaum, “Estructuras de datos en C y C++”. 2ª edición, Ed. Prentice Hall.

17. Levine G., “Programación estructurada y fundamentos de programación”, Mc Graw Hill, 2ª Ed., México, 1990.

18. Marzano R., “Dimensiones del aprendizaje”, Ediciones ITESO, México, 1992.

19. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. “A multi-national, multi-institutional study of assessment of programming skills of first-year CS students”, SIGCSE Bulletin, 33(4), pp. 125-180. 2001.

20. Peña Marí R., “Diseño de Programas: Formalismo y Abstracción”. Prentice Hall, México, 2005.

21. Peñaloza E., “Fundamentos de programación C/C++”, 4ª edición, Alfaomega, México, 2004.

22. Perry G., “Aprendiendo programación orientada a objetos con Turbo C++ en 21 días”. Ed. Prentice Hall Hispanoamericana, 1995.

23. Ponce, V.M, “Necesidad de articulación de teorías del aprendizaje”, Secretaría de Educación de Jalisco, Seminarios de formación e investigación educativa, México, 2006.

24. Pozo, J. I. “Aprendices y maestros”. Alianza Editorial, 2008.

25. Ricci A., “Fondamenti di Informatica LB – Modulo 01 Il Linguagio C. II”. Facolta’ di Ingegneria – Cesena, Universita’ degli Studi di Bologna, 2003.

26. Rudd A., C++ Complete, ed. Wiley-QED Publication 1994.

27. Robins A, Rountree J, Rountree N, “Learning and Teaching Programming: A Review and Discussion”. Computer Science Education, Vol 13, No 2, pp. 137-172, 2003.

28. Sánchez, M. A., Chamorro, F., Molina, J. M. y Mantellán, V. “Programación estructurada y fundamentos de programación”, Mc-Graw Hill, México, 1996.

29. Staugaard, Jr., “Técnicas estructuradas y orientadas a objetos”, Prentice-Hall, 2ª Edición, México, 1998.

30. Swan T., “Learning C++”, ed. SAMS 1991.

31. Terrence P., “Lenguajes de programación”, Ed. Prentice Hall, 1998.

32. Watt D. and Brown, D. “Java Collections, An introduction to Abstract Data Types, Data Structures and Algorithm”. John Wiley and Sons, New York, 2001.

33. Weiss M.A., “Data Structures and Algorithm Analysis in Java”. Addison Wesley, New York, 1999.

34. Weiss M. A., “Estructura de Datos y Algoritmos”. Addison-Wesley Iberoamericana, México, 1995.

35. Wertsch, J.V. “Vygotsky y la formación social de la mente” Paidós, 1988.

Inicialízate en la Programación con C++
Se terminó de imprimir el 11 de diciembre de 2013 en
Publidisa Mexicana S. A. de C.V.
Calz. Chabacano No. 69, Planta Alta
Col. Asturias C.P. 06850.
50 ejemplares en papel bond 90 gr.