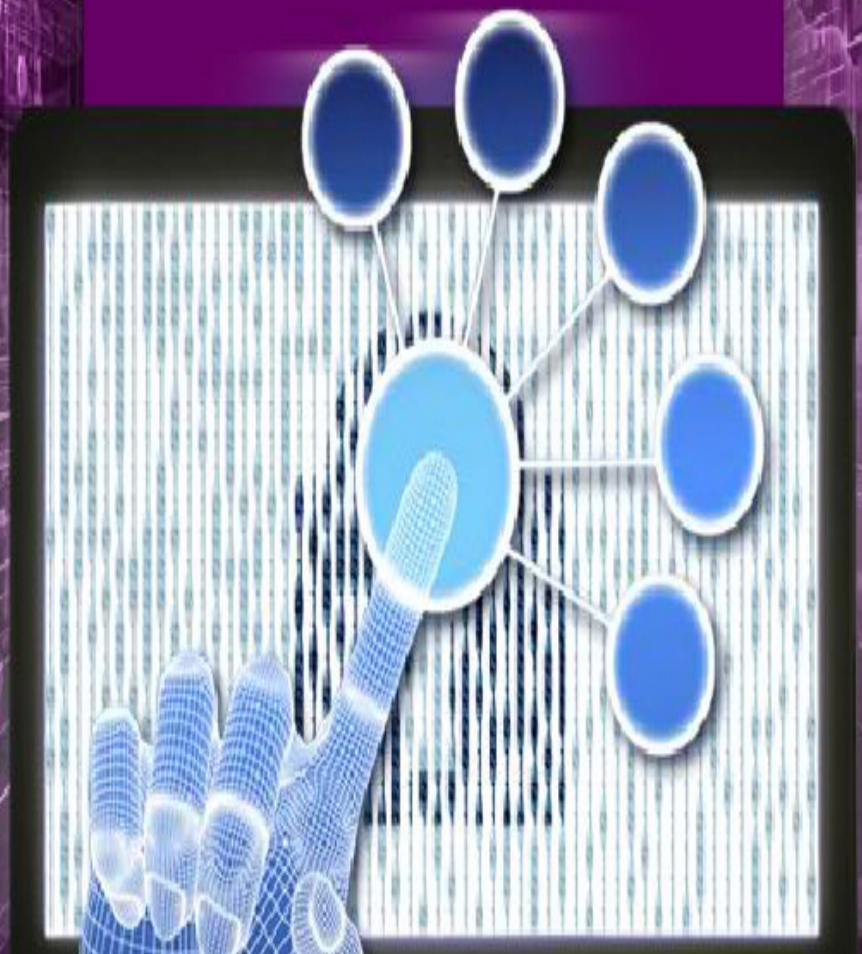
EDITORIAL DIGITAL
TECNOLÓGICO DE MONTERREY

FUNDAMENTOS DE PROGRAMACIÓN: UN ENFOQUE PRÁCTICO



SERGIO FRANCISCO
HERNÁNDEZ ALAMILLA

Autor
Editorial

Primera edición
De venta en: Amazon Kindle, Apple Books, Google Books y Amazon.
Fragmento editado, diseñado, publicado y distribuido por el Instituto Tecnológico y de Estudios Superiores de Monterrey.

Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849 |
Monterrey, Nuevo León | México.

Acerca de este eBook

El Tecnológico de Monterrey presenta su colección de eBooks de texto para programas de nivel preparatoria, profesional y posgrado. En cada título se integran conocimientos y habilidades que utilizan diversas tecnologías de apoyo al aprendizaje.

El objetivo principal de este sello es divulgar el conocimiento y experiencia didáctica de los profesores del Tecnológico de Monterrey a través del uso innovador de los recursos. Asimismo, apunta a contribuir a la creación de un modelo de publicación que integre en el formato de eBook, de manera creativa, las múltiples posibilidades que ofrecen las tecnologías digitales.

Con la Editorial Digital, el Tecnológico de Monterrey confirma su vocación emprendedora y su compromiso con la innovación educativa y tecnológica en beneficio del aprendizaje de los estudiantes dentro y fuera de la institución.

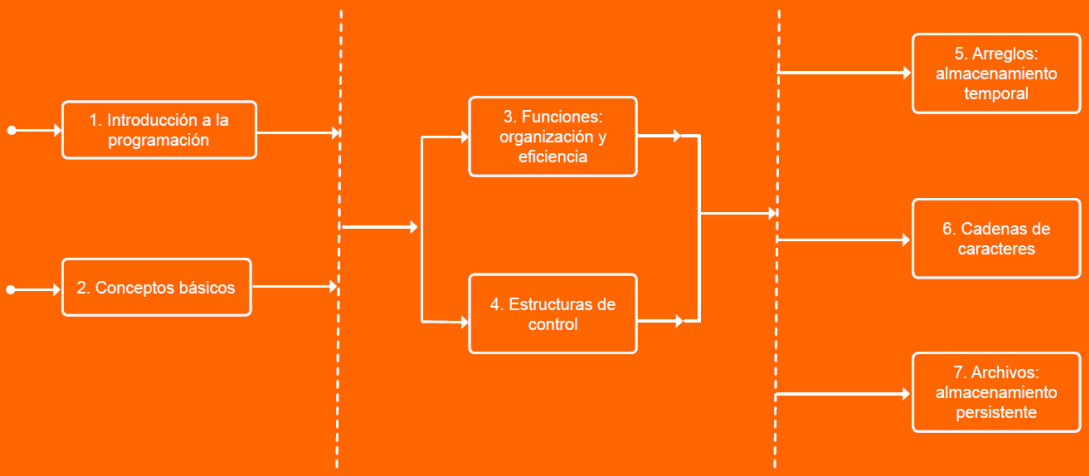
ebookstec@itesm.mx

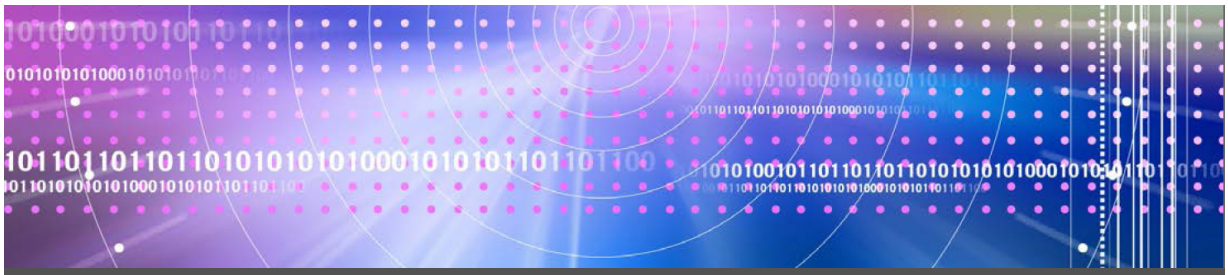
Acerca del autor

SERGIO FRANCISCO
HERNÁNDEZ ALAMILLA

Profesor del Tecnológico de Monterrey, Campus Cuernavaca, donde participa en proyectos de investigación y desarrollo en las áreas de gráficas computacionales e inteligencia artificial.Originario del estado de Veracruz, egresado de la Licenciatura en Ciencias Computacionales de la Universidad Autónoma de Puebla. En 2001 se incorporó a InovaWeb, empresa de desarrollo de software en la que participó como líder de proyectos de cobertura estatal y nacional para Ciba Geigi y Galia Textil. En 2004 cursó la Maestría en Ciencias Computacionales con especialidad en Inteligencia Artificial en el Tecnológico de Monterrey Campus Cuernavaca, donde participó como consultor en proyectos de minería de datos y gráficas computacionales para la empresa Tenaris Tamsa en conjunto con el INAOEP. Ha publicado dos artículos en congresos internacionales y participado en las expociencias de México y Francia. Recientemente se graduó de la Maestría en Animación y Arte Digital impartida por la Universidad Politécnica de Cataluña.

Mapa de contenidos





Introducción del eBook

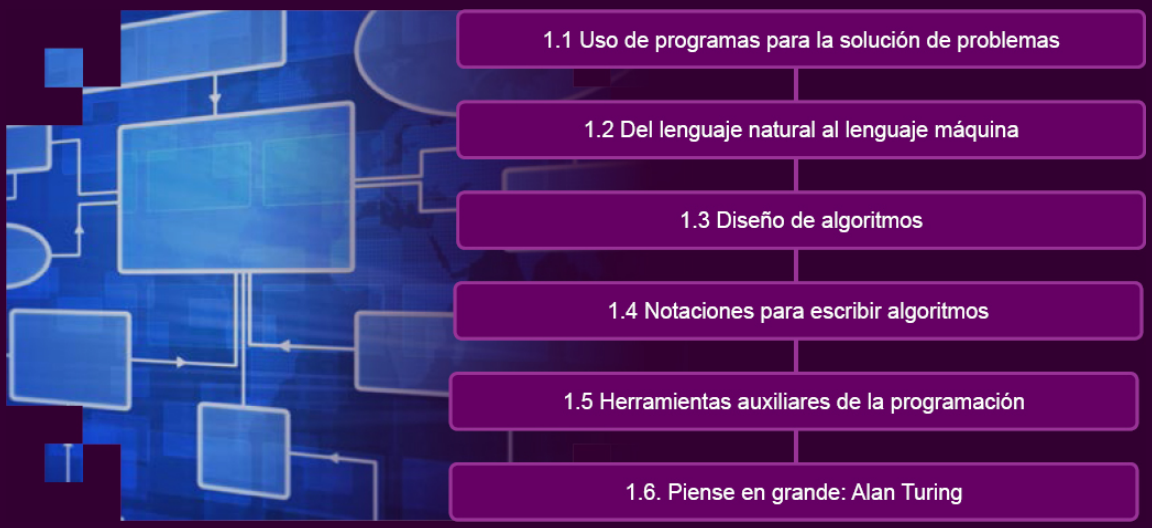
En los años recientes las aplicaciones de la computación se han diversificado rápidamente, ahora incursionan en áreas en las que jamás se habría pensado que podía intervenir una computadora. Las herramientas de software que se han generado son cada vez más sofisticadas. Sin embargo, los problemas actuales presentan retos aún mayores, que exigen de las herramientas computacionales el más alto grado de flexibilidad.

En consecuencia, día a día se presentan lenguajes de programación o **scripting**, que permiten extender la funcionalidad del software, e incluso crear nuevas herramientas de acuerdo con las necesidades de un proyecto. Es así como la programación ha dejado de ser un conocimiento exclusivo del desarrollador de software y se ha convertido en una poderosa herramienta para otros profesionistas: artistas digitales, diseñadores, ingenieros y financieros, entre otros.

Sin importar el dominio específico donde se utilice, los fundamentos de la programación son esencialmente los mismos, y su correcta comprensión es un primer paso a considerar. Este eBook aborda esos fundamentos desde un punto de vista práctico, apoyándose en ejemplos y ejercicios que permiten la adecuada comprensión de cada tema. Así, abre al lector una puerta a la enorme gama de oportunidades que ofrece el apasionante mundo de la programación.

Capítulo 1. Introducción a la programación

Organizador temático



Introducción a la programación

1.1. Uso de programas para la solución de problemas

Una de las competencias más valiosas en cualquier actividad profesional es la de analizar problemas e identificar soluciones eficientes que los resuelvan. En áreas donde el software juega un papel importante, esta labor es aún más compleja, ya que con frecuencia existen múltiples caminos para llegar a resultados similares. Por esta razón resulta de gran importancia comprender el funcionamiento del software que se utiliza día con día, para posteriormente adentrarse en el proceso que se sigue para crearlo.

En la Figura 1.1 se observa el funcionamiento general del software. Por principio éste recibe información de entrada, es decir, aquellos datos necesarios para su funcionamiento. Comúnmente esta información de entrada proviene de un archivo, del teclado, del *mouse*, de una **URL** o de un flujo de datos. El software procesa la información de entrada y ofrece información de salida. Según el propósito con que haya sido diseñado, la salida puede ser un mensaje en pantalla, una gráfica, una imagen, un archivo o cualquier otra representación de información válida.

Figura 1. 1. Funcionamiento general del software.

En la práctica, con frecuencia, pueden encontrarse herramientas de software prediseñadas que resuelven los problemas más comunes para un problema específico. Si se prueban estas herramientas en diferentes escenarios en la gran mayoría de los casos se resolverá el problema satisfactoriamente. Pero si se piensa más allá, ¿qué sucede cuando estas herramientas, no cuentan con la funcionalidad para hacer lo que un proyecto en particular requiere? O peor aún, ¿qué pasa si no existe una herramienta que ofrezca una solución al problema que se ha presentado?

Sin importar cuán sofisticada sea una pieza de software, siempre habrá situaciones de uso no contempladas en su diseño original que no resolverá.

En este punto la programación entra en escena ya que permite agregar funcionalidad a las herramientas existentes, o bien, diseñar nuevos elementos de software.

Elaborar un programa, en términos prácticos, consiste en escribir una secuencia de instrucciones en un lenguaje que la computadora sea capaz de interpretar y procesar.

La programación es el proceso que consiste en escribir, analizar y depurar programas, diseñados en un lenguaje de programación, para que sean interpretados correctamente por una computadora.

Por si esto fuera poco, de manera colateral, la programación desarrolla en el programador habilidades muy útiles, altamente valoradas y difíciles de desarrollar como la capacidad de abstracción, de análisis, de síntesis y el razonamiento lógico. Además, la programación permite automatizar tareas repetitivas y operaciones complejas, cuya realización demandaría mucho más tiempo y precisión del que puede disponer una persona.

1.2. Del lenguaje natural al lenguaje máquina

Comunicar instrucciones de manera efectiva a una computadora no es una tarea fácil, de hecho ha sido un tema estudiado durante décadas por expertos en ciencias computacionales. La principal dificultad asociada a este problema radica en los mecanismos de comunicación tan disímiles que utilizan las personas y las computadoras. Mientras las primeras se expresan a través del uso su lenguaje natural, las segundas entienden a la perfección el simple y elemental código binario.

Cada una de estas formas de comunicación tiene ventajas y desventajas. El lenguaje natural, por un lado, es adecuado para las personas por su riqueza expresiva, pero tiene la enorme desventaja de ser ambiguo y demasiado impreciso para ser interpretado correctamente por una computadora. Por su parte, el código binario es preciso y concreto, lo que facilita que la computadora lo comprenda, pero es muy difícil de utilizar por las personas, debido sobre todo a su escasa capacidad expresiva y a lo complejo que resulta asociar cada significado con un término o símbolo numérico.

Como es de suponerse ante tal desbalance el punto medio es con frecuencia lo más adecuado. Partiendo de este razonamiento, se han creado decenas, quizá cientos, de lenguajes de programación, que no son más que una representación intermedia entre el lenguaje natural de las personas y el lenguaje binario de la computadora. En la Figura 1.2 puede observarse este fenómeno.

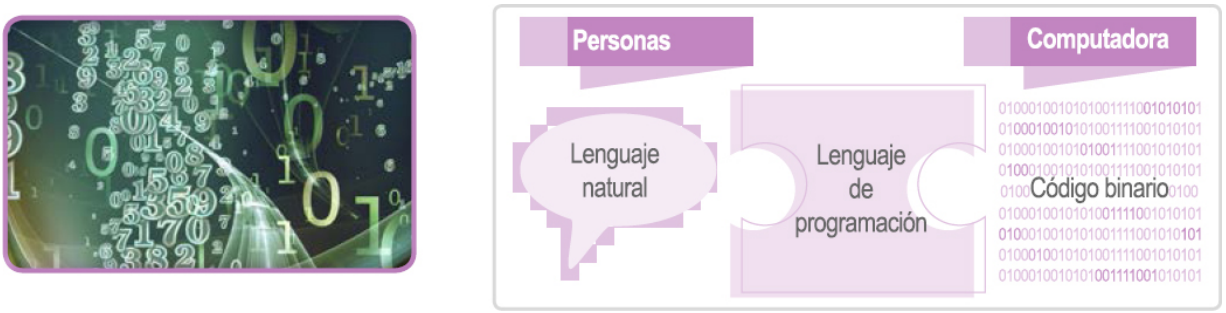


Figura 1. 2. Diferentes mecanismos de comunicación.

El lenguaje de programación es un conjunto de reglas sintácticas y semánticas que definen la forma de escribir programas para ser interpretados por una computadora.

Los lenguajes de programación buscan proporcionar al usuario lo mejor de ambos mundos. Por un lado, tienen una sintaxis bien definida que les permite ser expresados sin ambigüedad y, al mismo tiempo, su semántica ofrece la flexibilidad y la expresividad necesarias para representar soluciones a problemas complejos.

Cuando se usa un lenguaje de programación es posible comunicar instrucciones a la computadora de manera cada vez más eficiente y precisa, con el fin aprovechar al máximo sus ventajas en favor de la generación de soluciones a los principales problemas de la industria y de las personas.

1.3. Diseño de algoritmos

Hasta este punto se ha abordado el funcionamiento general del software y la dificultad implícita de programar una computadora. Sin embargo, es importante no perder de vista que la solución al

problema es diseñada por las personas. Para ello primero se lleva a cabo un análisis concienzudo del problema, luego se elabora una secuencia ordenada de los pasos que deben llevarse a cabo para resolverlo y finalmente se elabora el programa, es decir, se traduce la solución que se diseñó a un lenguaje de programación que la computadora sea capaz de entender.

A la secuencia ordenada de pasos que deben seguirse para resolver un problema se le denomina algoritmo.

Un algoritmo debe ser en principio: preciso, definido y finito. Su diseño se deriva directamente del análisis que se hace del problema, y es independiente de cualquier lenguaje de programación.

En la Figura 1.3 se observa el proceso de programación y la relación entre cada una de las actividades que lo integran. Es muy importante notar la separación entre la fase de resolución del problema y la fase de trabajo en la computadora. Un principio primordial para la programación eficiente consiste en que el algoritmo deberá ser diseñado antes de escribir una sola instrucción en la computadora. Esta es la regla de oro que se debe recordar y privilegiar siempre que se emprenda la tarea de programar una computadora.

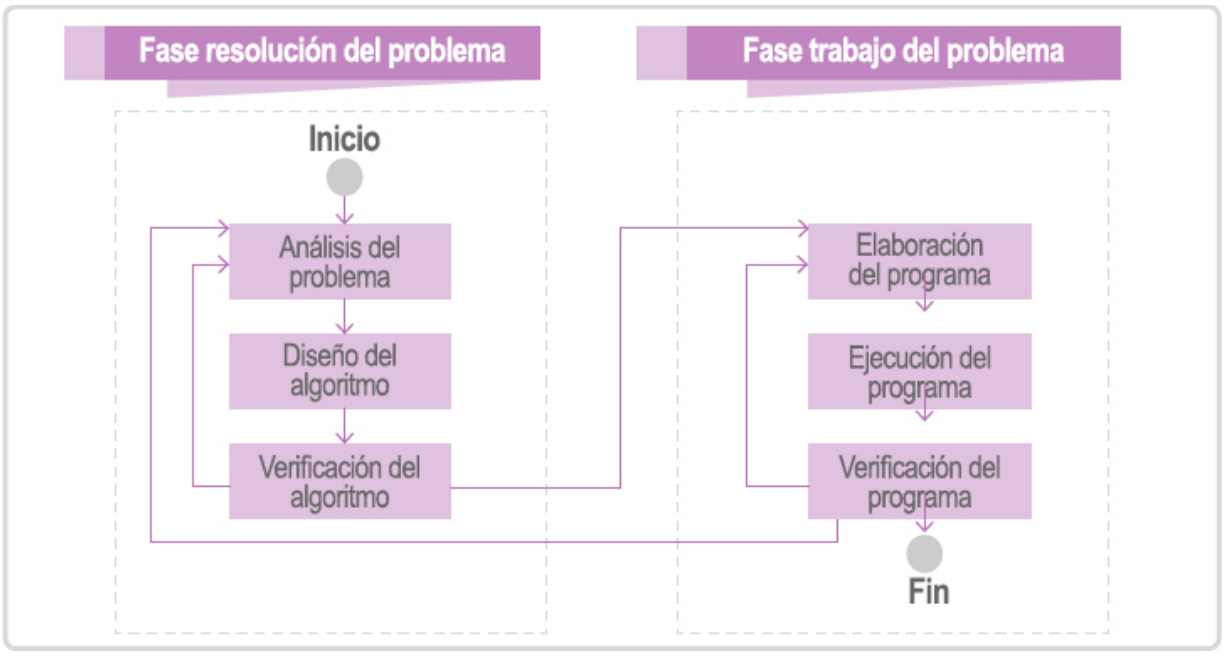
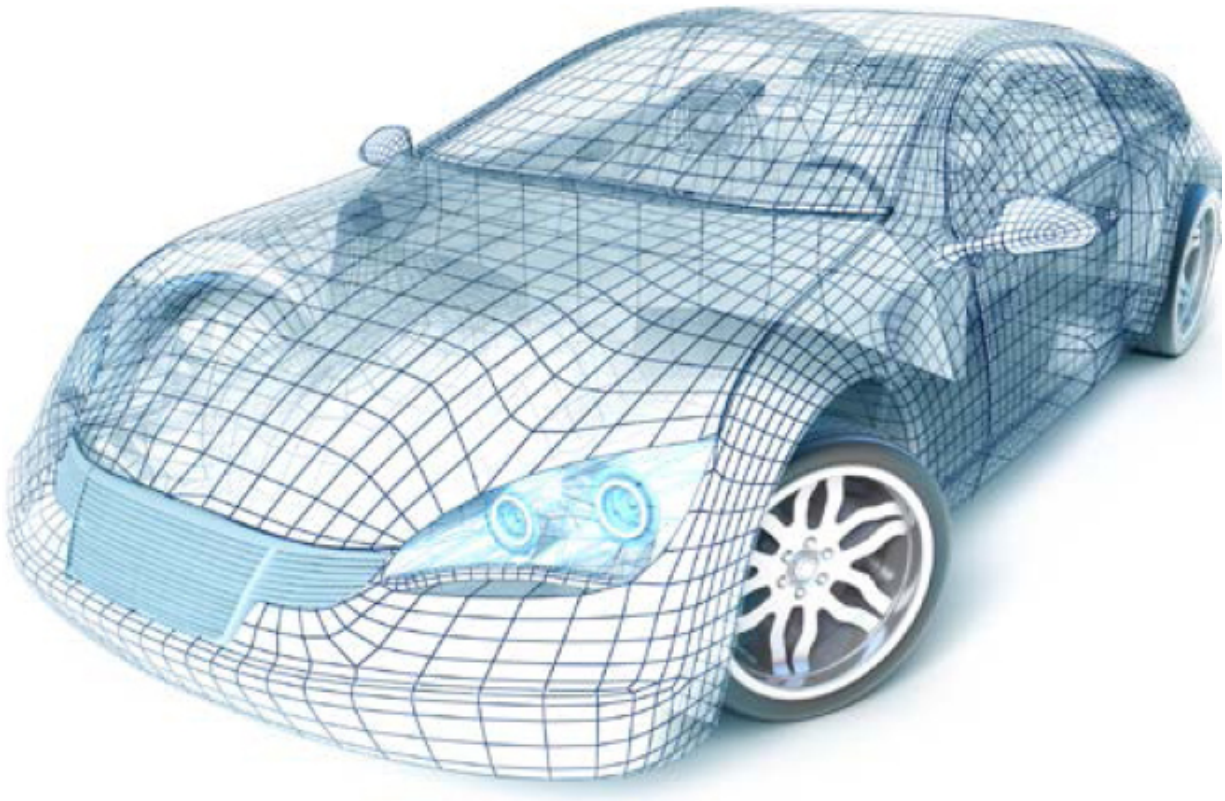


Figura 1. 3. Proceso para la solución de problemas a través de la programación.

A continuación se muestra un ejemplo concreto que permitirá comprender mejor el proceso descrito:



Enunciado del problema: Actualmente en la planta automotriz se están realizando pruebas con los nuevos modelos de autos, una de ellas consiste en determinar el rendimiento de gasolina de manera experimental. Para ello se conduce el auto durante una cierta distancia y se registran la distancia recorrida en kilómetros y cantidad precisa de gasolina empleada en litros.

Análisis del problema

El primer paso consiste en identificar los datos de entrada y los datos de salida (ver la Figura 1.1). Además es necesario distinguir aquellos elementos importantes de los que no son relevantes dentro del enunciado del problema.

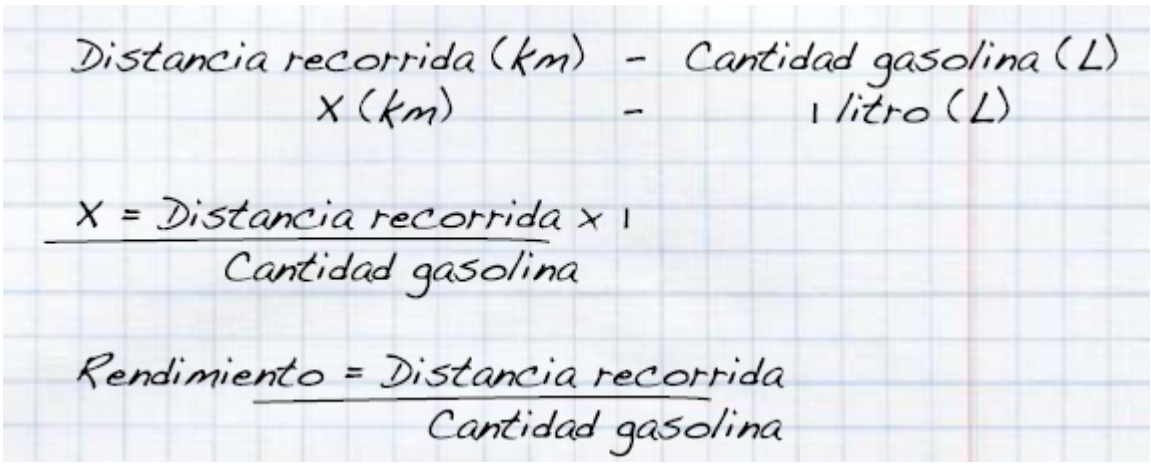
Datos de entrada: Distancia recorrida (en kilómetros) y cantidad de gasolina (en litros).

Datos de salida: Rendimiento de gasolina (en kilómetros por litro).

Análisis: El dato de que se el auto se desarrolla en una planta automotriz es irrelevante, lo mismo el dato que el auto es nuevo. El enunciado podría quedar de la siguiente manera:

“Dados los datos de la distancia recorrida y la cantidad de gasolina empleada, determinar el rendimiento de un auto en kilómetros por litro”.

Al explorar las fórmulas o relaciones útiles, se plantea la solución utilizando una regla de tres que involucre los datos de los que se disponen:



Dado el análisis, las variables necesarias para elaborar el algoritmo serían: distancia, gasolina y rendimiento.

Diseño del algoritmo

El segundo paso consiste el escribir el algoritmo. En la Figura 1.4 se observa el algoritmo diseñado; siempre se comienza por Inicio y se avanza de manera secuencial hasta terminar en Fin. El diseñador de algoritmos debe tener como objetivo cumplir con tres importantes características: que el algoritmo sea preciso, sea definido y sea finito.

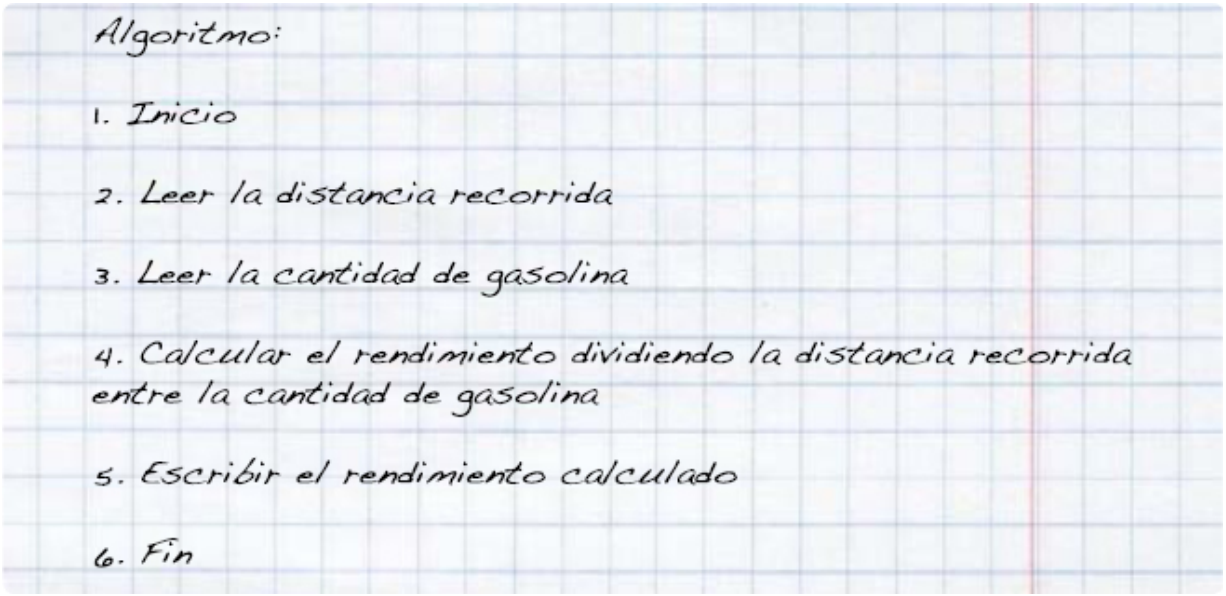


Figura 1. 4. Algoritmo diseñado para la solución del problema.

Verificación del algoritmo

El tercer paso consiste en validar que el algoritmo en efecto resuelva el problema que se planteó. Comúnmente se utilizan casos de prueba conocidos u operaciones con resultados predecibles. Se debe pensar en cubrir el mayor espectro posible de datos de entrada. En la Figura 1.5 se observa el uso de un caso específico para probar el algoritmo. Según los resultados obtenidos, es factible regresar al análisis del problema, modificar el algoritmo y verificar las veces que sea necesario.

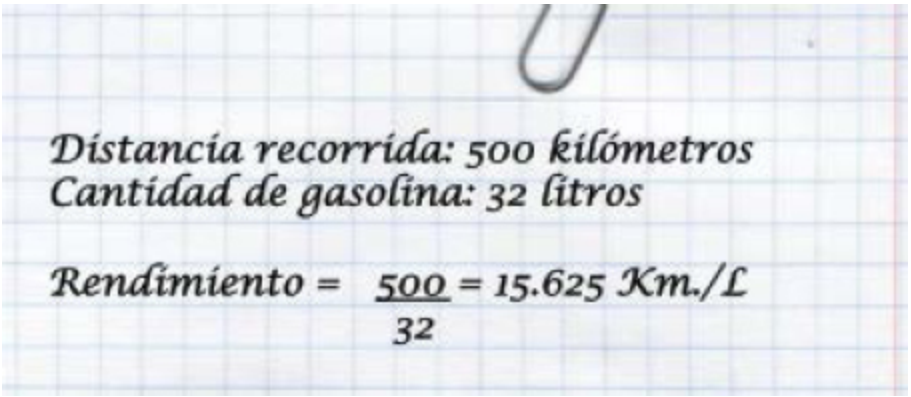
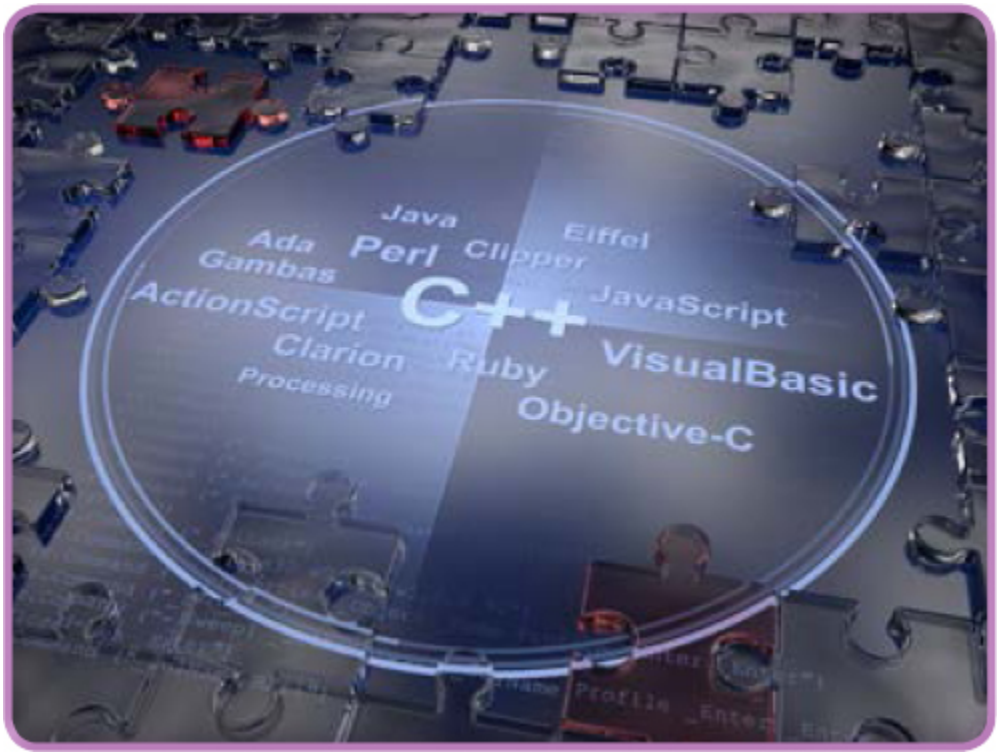


Figura 1. 5. Verificación de un algoritmo.

Se sugiere al lector no abandonar la fase de resolución del problema hasta no tener un alto nivel de certeza sobre el algoritmo diseñado. Invertir tiempo en esta fase, ahorra mucho tiempo en etapas posteriores del proceso.

Elaboración del programa

Una vez diseñado el algoritmo que resuelve el problema, y que éste fue verificado, se traduce a un lenguaje de programación. En la Figura 1.6 aparece el algoritmo traducido al **lenguaje de programación C++**, que se usará a lo largo de este eBook. No se espera que en este momento el lector identifique cada una de las partes del programa, ni mucho menos las reglas que se siguieron para su escritura. El programa se muestra al lector con fines ilustrativos, con la intención de ubicar el momento de su creación dentro del proceso general. En los siguientes capítulos se abordará a detalle el tema del lenguaje de programación.



```
#include <iostream>

using namespace std;

int main(){
  double distancia, gasolina, rendimiento;

  cout << "Indique distancia recorrida: ";
  cin >> distancia;

  cout << "Indique gasolina empleada: ";
  cin >> gasolina;

  rendimiento = distancia / gasolina;

  cout << "El rendimiento es " << rendimiento << " km/lt" << endl;

  return 0;
}
```

Figura 1. 6. Programa escrito en el lenguaje de programación C++ que calcula el rendimiento de un auto.

Nótese que al traducir el algoritmo al lenguaje de programación, el número de instrucciones es mayor al número de pasos del mismo algoritmo expresado en lenguaje natural. Esto era de esperarse, ya que el mayor nivel de expresividad lo posee indiscutiblemente el lenguaje natural. Sin embargo, es justo también aclarar que el mismo algoritmo expresado en código binario, el lenguaje computacional, tomaría varias decenas de líneas. De ahí el que los lenguajes de programación se hayan consolidado desde hace varios años, como el paso intermedio entre el lenguaje de las personas y el de la computadora.

Verificación del programa

Esta tarea consiste en corroborar que el programa funcione correctamente, no sólo con los ejemplos utilizados en su diseño, sino bajo cualquier circunstancia. Para ello se sugiere analizar los

posibles escenarios de uso y alimentar con ellos el programa para validar su funcionamiento.

Del conjunto de pruebas a las que se somete el programa pueden derivarse modificaciones al mismo, como añadir instrucciones para mejorar su funcionamiento e incluso modificar la estructura del programa. Por este motivo es muy importante que esta tarea se realice con sumo cuidado. Incluso puede darse el caso que estas pruebas revelen algún defecto grave en el diseño del algoritmo, lo cual obligaría a regresar a la fase de resolución del problema. Esta situación, aunque poco frecuente, está contemplada en la Figura 1.3.

Lo más importante del proceso descrito es separar la tarea de diseñar y verificar el algoritmo de la tarea de escribir y verificar el programa. Estas dos tareas son ciertamente complementarias, pero de naturaleza muy distinta. Por un lado, el diseño de un algoritmo se centra en encontrar la solución al problema, mientras que la escritura de un programa se centra en traducir un algoritmo a un lenguaje de programación, de acuerdo con las reglas del mismo.

Ejemplos:

A continuación se presentan algunos ejemplos en los que se observa cada etapa de la fase de resolución del problema. Es primordial que esta primera parte quede muy clara para el lector, de tal forma que facilite su aprendizaje en los temas siguientes.

En la figura 1.7 puede revisarse, detalladamente, el proceso a seguir en un nuevo problema, el de las patinetas:

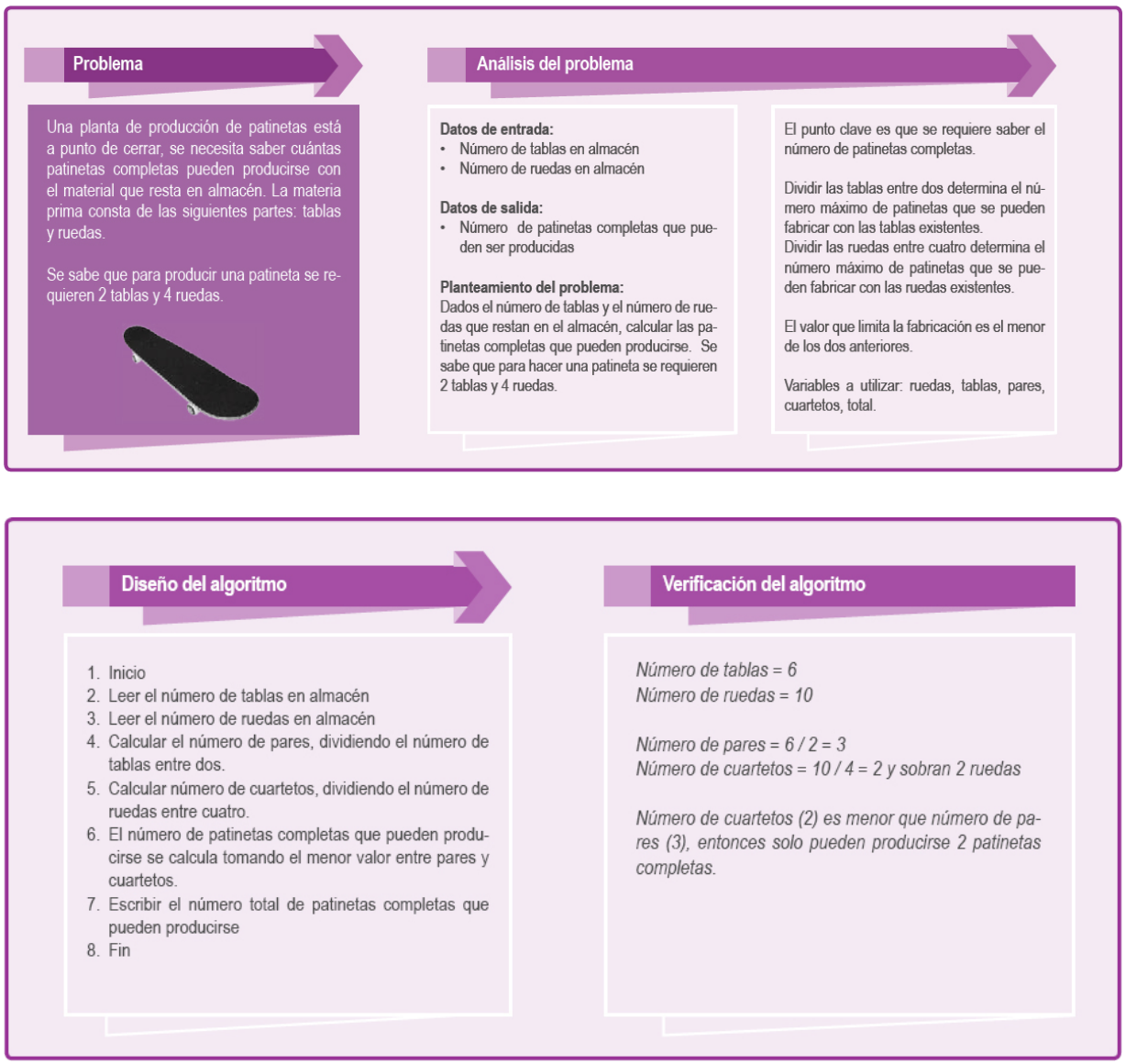


Figura 1. 7. Fase de resolución del problema para el problema de las patinetas.

Observe cómo de cada una de las etapas se debe derivar, naturalmente, la siguiente. Es importante que si el lector se está iniciando en el mundo de la programación, lleve a cabo con cuidado cada etapa. Poco a poco, algunas etapas irán fluyendo de manera más natural conforme se adquiere habilidad en la solución de problemas.

De la misma forma, ahora se revisará otro problema, el de la detección de colisiones. En la Figura 1.8 pueden apreciarse cada una de las etapas de la fase de resolución del problema.

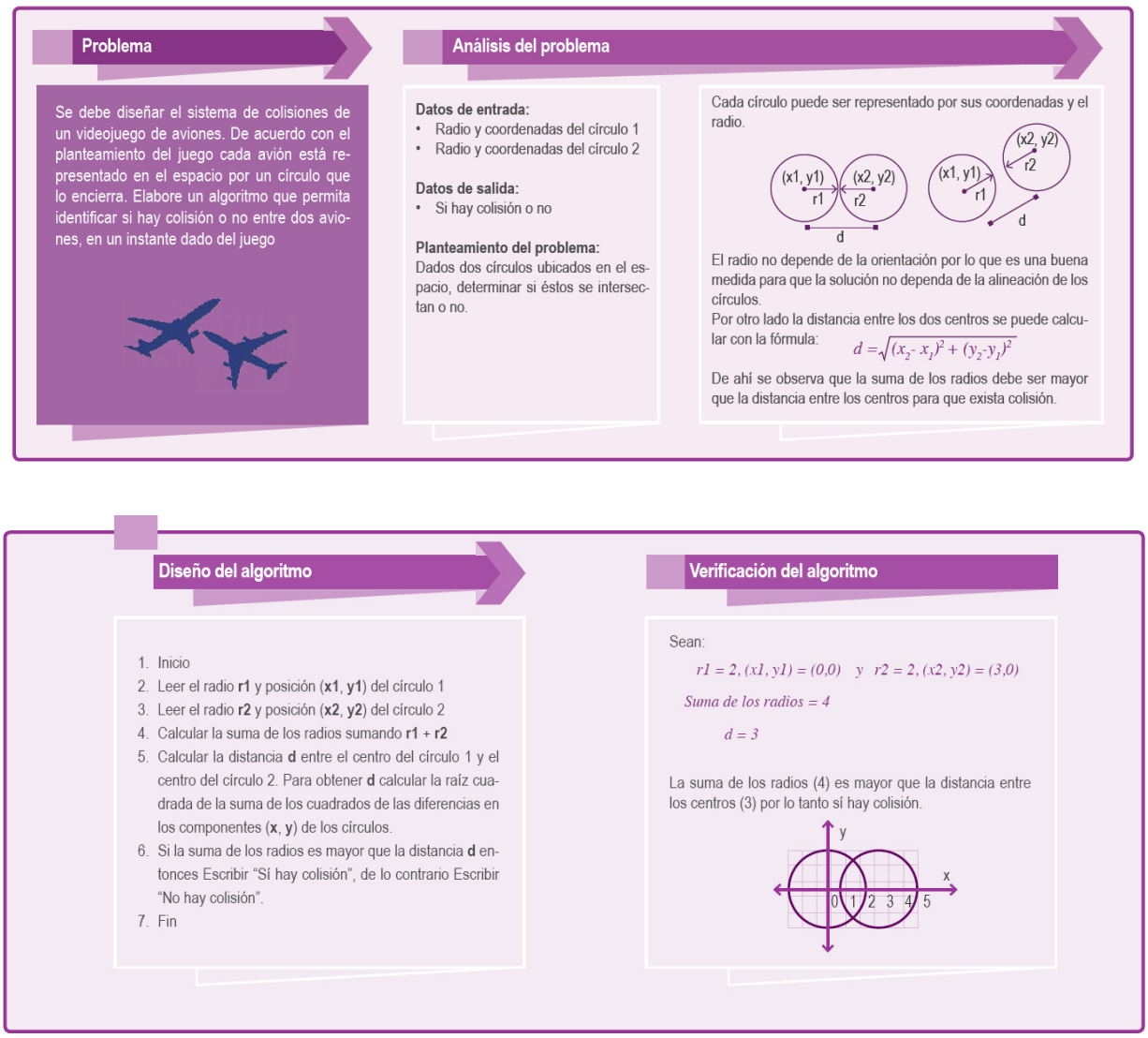


Figura 1. 8. Fase para la resolución del problema de las colisiones.

1.4. Notaciones para escribir algoritmos

El uso del lenguaje natural no es la única opción al momento de escribir un algoritmo. Existen además otras opciones, cada una con ventajas y desventajas respecto de las demás. En este

eBook se abordarán las tres más utilizadas: lenguaje natural, diagramas de flujo y pseudocódigo.

Lenguaje natural: Esta opción se presenta como la más accesible, su capacidad expresiva es muy alta, pero con frecuencia tiende a ser ambigua. Por otro lado, en el caso de problemas más complejos, resultaría muy complicado describir cada detalle del algoritmo en lenguaje natural.

Diagramas de flujo: Esta forma de representar algoritmos, involucra elementos gráficos para representar las operaciones que se pueden realizar. Cada elemento tiene un significado determinado y está unido a los demás por flechas que especifican el orden en que se realiza la secuencia. En la Figura 1.9 pueden observarse los principales símbolos gráficos utilizados en los diagramas de flujo.

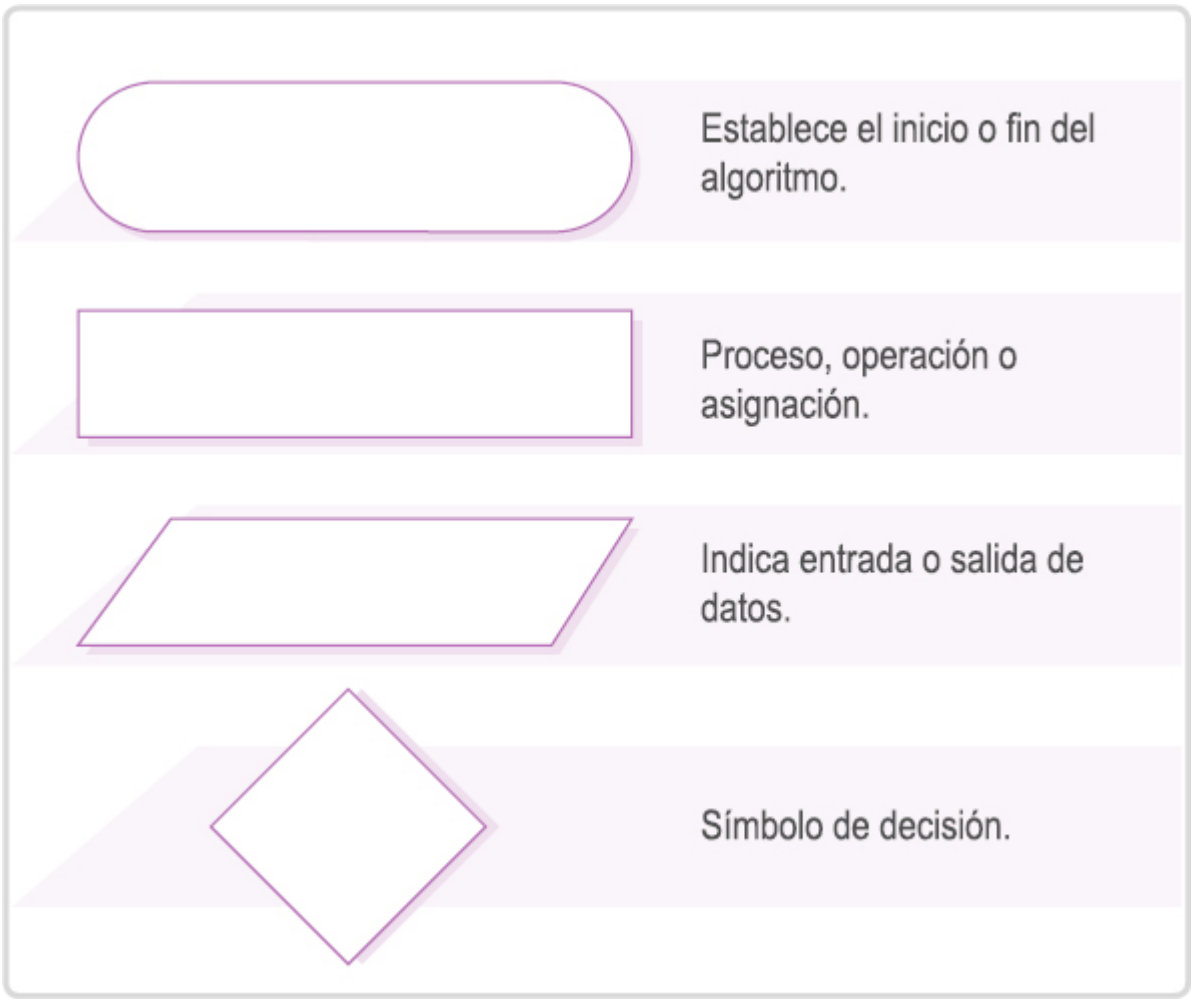


Figura 1. 9. Simbología utilizada para construir diagramas de flujo.

En los ejemplos mostrados en la Figura 1.9 puede observarse que aunque se utiliza el mismo símbolo para entrada y salida de datos, son las palabras Obtener y Mostrar las que determinan el uso específico del símbolo en el contexto de un algoritmo. También se usan las palabras leer y escribir para denotar entrada y salida de datos, respectivamente.

Cada algoritmo, representado por medio de un diagrama de flujo, debe cumplir con ciertas reglas básicas de diseño:

1. Por principio todo algoritmo debe tener un solo símbolo de inicio y uno de fin.
2. Después de realizar cada acción, debe quedar claro cuál es el siguiente paso.

3. Todos los posibles caminos deben conducir, eventualmente, al fin del algoritmo.

En la Figura 1.10 se observan los diagramas de flujo que corresponden a los algoritmos para calcular el rendimiento de gasolina y la producción de patinetas completas.

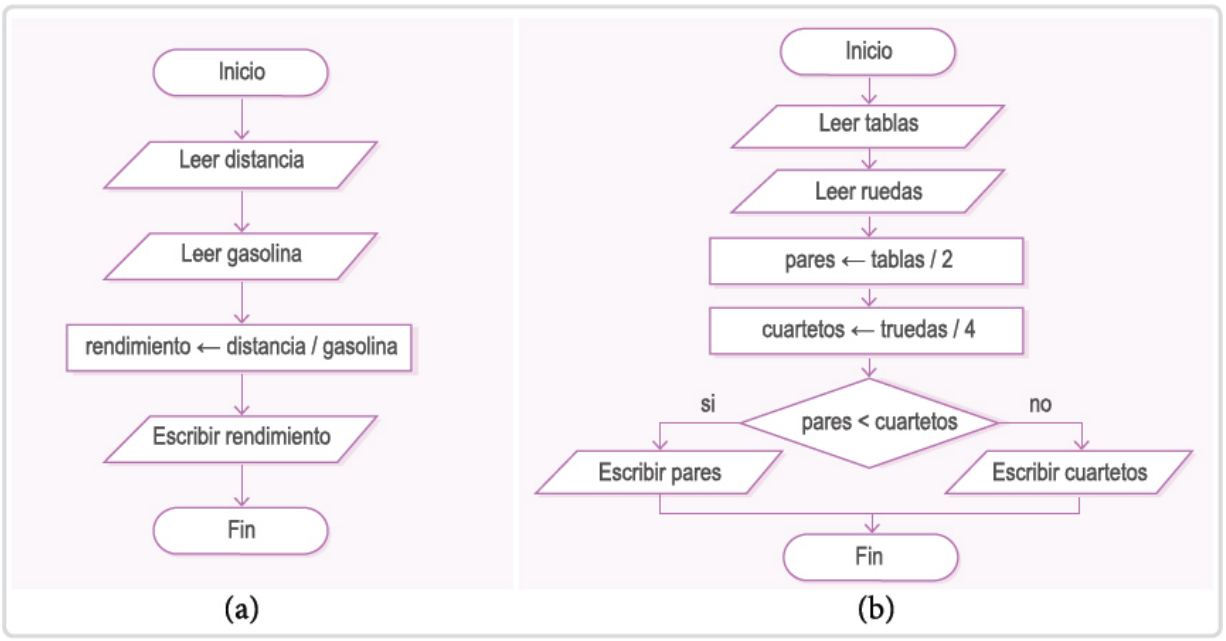


Figura 1. 10. Algoritmos expresados en forma de diagrama de flujo. (a) Para calcular el rendimiento de gasolina. (b) Para calcular la producción máxima de patinetas completas.

La representación que se muestra en la Figura 1.10 resulta conveniente para aquellas personas que comprenden los conceptos de manera visual, ya que está basada en elementos gráficos. De

hecho es muy común que quienes se inician en el mundo de la programación, encuentren esta representación más comprensible, pues presenta con claridad la secuencia y el funcionamiento del un algoritmo, paso a paso. Las bifurcaciones en el flujo del algoritmo, como la que se observa en la Figura 1.10 (b), son fácilmente comprensibles bajo esta notación ya que resulta muy claro que el flujo continúa por uno de los dos posibles caminos.

Si el lector desea experimentar con esta notación para la representación de algoritmos hay diferentes herramientas diseñadas para ese fin. *Raptor* –por ejemplo– está diseñada para elaborar y ejecutar diagramas de flujo que proporciona un acercamiento directo al diseño de algoritmo de manera visual, simulando incluso el algoritmo en ejecución.

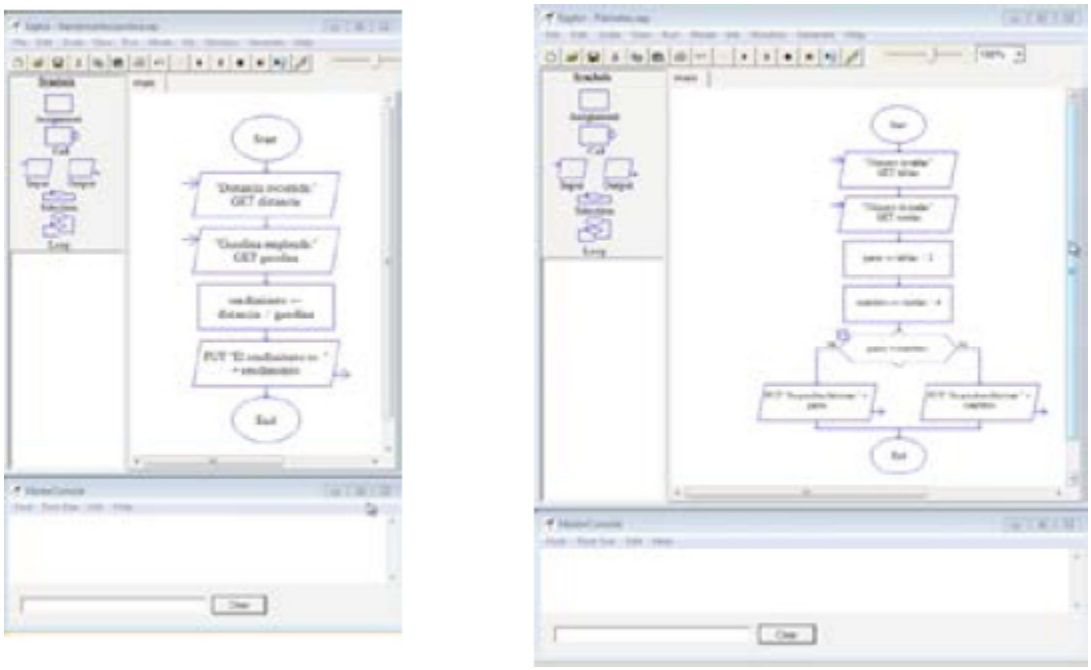


Figura 1. 11. Diagramas de flujo elaborados en Raptor. (a) Para calcular el rendimiento de gasolina. (b) Para calcular la producción máxima de patinetas completas.

Cabe aclarar que hay variaciones en la notación utilizada por Raptor, esto debido a las convenciones de la propia herramienta. Por ejemplo, emplea las palabras GET y PUT en lugar de leer y escribir. Además, agrega mensajes de texto adicionales a las entradas y salidas. Por otro lado, añade pequeñas flechas a la

izquierda o derecha del símbolo de entrada/salida de datos con la finalidad de hacer más evidente la diferencia entre un uso y el otro. Finalmente, Raptor amplía en símbolo de decisión por cuestiones de despliegue en pantalla, ya que por ser una forma romboidal su tamaño se incrementaría desproporcionadamente para expresiones de gran tamaño.

Pseudocódigo: Esta representación es la más cercana a lo que significaría escribir un programa directamente en un lenguaje de programación. Su sintaxis es menos específica que la de un lenguaje de programación, pero mantiene una estructura muy similar. En la Figura 1.12 se muestra la notación básica para el uso del pseudocódigo: (los elementos delimitados por los símbolos <> denotan elementos genéricos y deben ser sustituidos por símbolos específicos dentro los algoritmos que se diseñen).

`Inicio`
 	
`Fin`
	Establecen el inicio o fin del algoritmo.

`<expresión>`	Proceso, operación o asignación.

`Leer <variable>`
 	
`Escribir <variable>`
	Indica entrada o salida de datos.

`Si (<expresión_lógica>)`
`Entonces`
`  // instrucciones`
`Sino`
`  // instrucciones`
`Finsi`
	Establece una decisión basada en una expresión lógica.

Figura 1. 12. Notación utilizada para escribir algoritmos en pseudocódigo.

Normalmente esta representación se utiliza por quienes están muy familiarizados con el diseño de algoritmos, y es una representación estándar para comunicar algoritmos de manera efectiva y específicas, pero sin entrar en la sintaxis de un lenguaje de programación en particular. En la Figura 1.13 se observan los algoritmos de rendimiento de gasolina y producción de patinetas representados en forma de pseudocódigo.

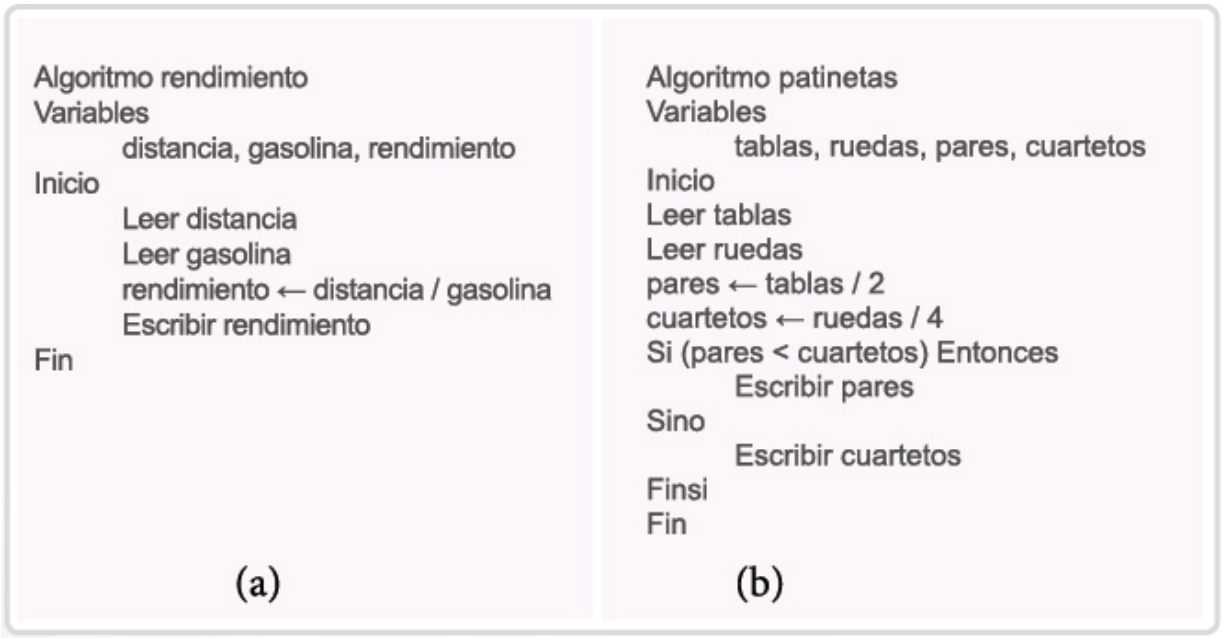


Figura 1. 13. Algoritmos representados en forma de pseudocódigo. (a) Para calcular el rendimiento de gasolina. (b) Para determinar la producción máxima de las patinetas.

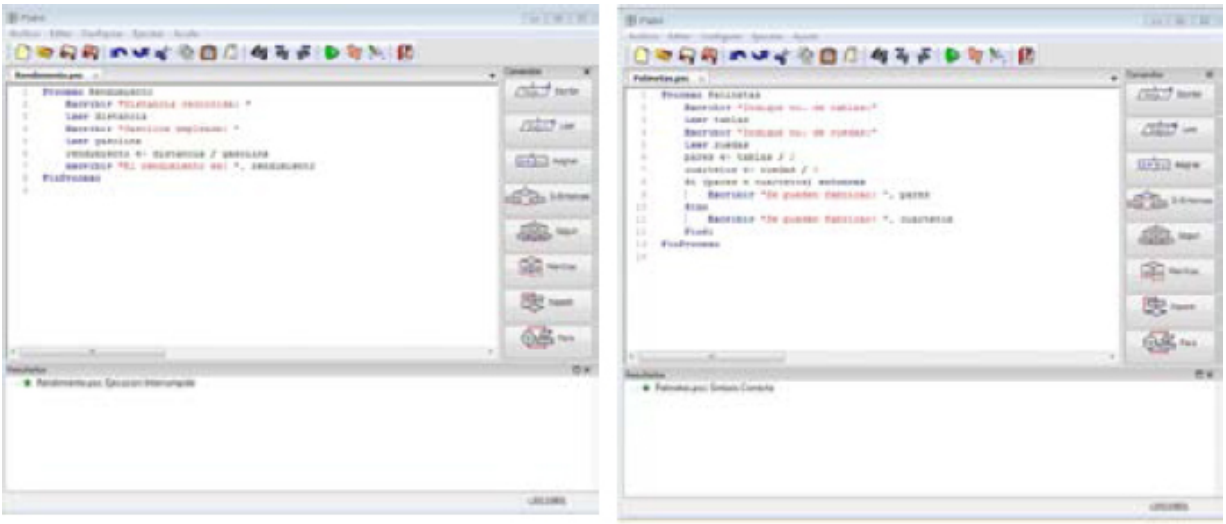
Como se observa en los ejemplos, para la especificación de algoritmos en términos de pseudocódigo existe un esquema general. Éste se muestra a continuación:

```
Algoritmo <nombre_del_algoritmo>
   [  Constantes
       <declaraciones_de_constantes> ]
   [  Tipos_de_datos
       <declaraciones_de_tipos_de_datos> ]
   [  Variables
       <declaraciones_de_variables> ]
```

```
Inicio
    <bloque_de_instrucciones>
Fin
```

Hasta el momento, sólo se han identificado a las variables como contenedores que representan a los datos con los que opera el programa. En el capítulo 2 se abordará este tema con mayor detalle, además de las constantes y los tipos de datos. Estos dos últimos términos aparecen en el esquema general que se presenta y es posible que en este momento quizá escapen a la comprensión del lector.

Si el lector desea experimentar con esta notación para la representación de algoritmos existen múltiples herramientas diseñadas para ese fin. *PseInt,* por ejemplo, es una herramienta diseñada para elaborar e interpretar pseudocódigos.



(a) (b)

Figura 1. 14. Pseudocódigo elaborado en PseInt. (a) Para calcular el rendimiento de gasolina. (b) Para calcular la producción máxima de patinetas completas.

Sin importar cuál sea la representación del algoritmo en cuestión, lo más importante es escribirlo de la manera más precisa y clara posible. En este eBook se utilizará mayormente el pseudocódigo

debido a su capacidad expresiva y a la expansión que ha tenido su uso en el mundo de la programación y el *scripting.*

Una vez que se analiza el problema y se diseña el algoritmo que lo resuelve, es momento de proceder a escribir el programa, para lo cual se requieren herramientas especializadas. En la siguiente sección se abordará el tema de las herramientas y el software necesario para escribir programas en un lenguaje de programación específico.

1.5. Herramientas auxiliares de la programación

Así como existe software diseñado para redactar documentos, para procesar imágenes y para navegar en internet, también hay software diseñado para escribir programas. El objetivo de este software es proporcionar a las personas un ambiente cómodo y las herramientas necesarias para hacerlo.

Lo primero que se requiere para escribir un programa es un editor de texto. Éste puede ser el editor básico que todo sistema operativo tiene, o bien el editor más sofisticado que los ambientes integrados de desarrollo, IDE por sus siglas en inglés, proporcionan. En cualquier caso el editor es necesario, ya que es ahí donde se escriben las instrucciones. A este conjunto de instrucciones de programación se le denomina comúnmente **código fuente**.

A un programa, compuesto por una serie de instrucciones, derivado de un algoritmo y escrito en un lenguaje de programación, se le denomina código fuente.

Hasta este punto, el código fuente no es comprensible directamente para la computadora; cabe recordar que el lenguaje de programación es una representación intermedia. Para transformar el código fuente en una representación asequible, es necesario un traductor.

Primero se traduce el código fuente a **código objeto** y luego, éste último se traduce en **código ejecutable**. Estas dos tareas se realizan por dos programas: el **compilador** y el **enlazador**, respectivamente.

Como parte de esta traducción, el compilador revisa que el programa esté correctamente escrito, de acuerdo con las reglas del lenguaje de programación. En la Figura 1.14 se observa el procedimiento descrito.

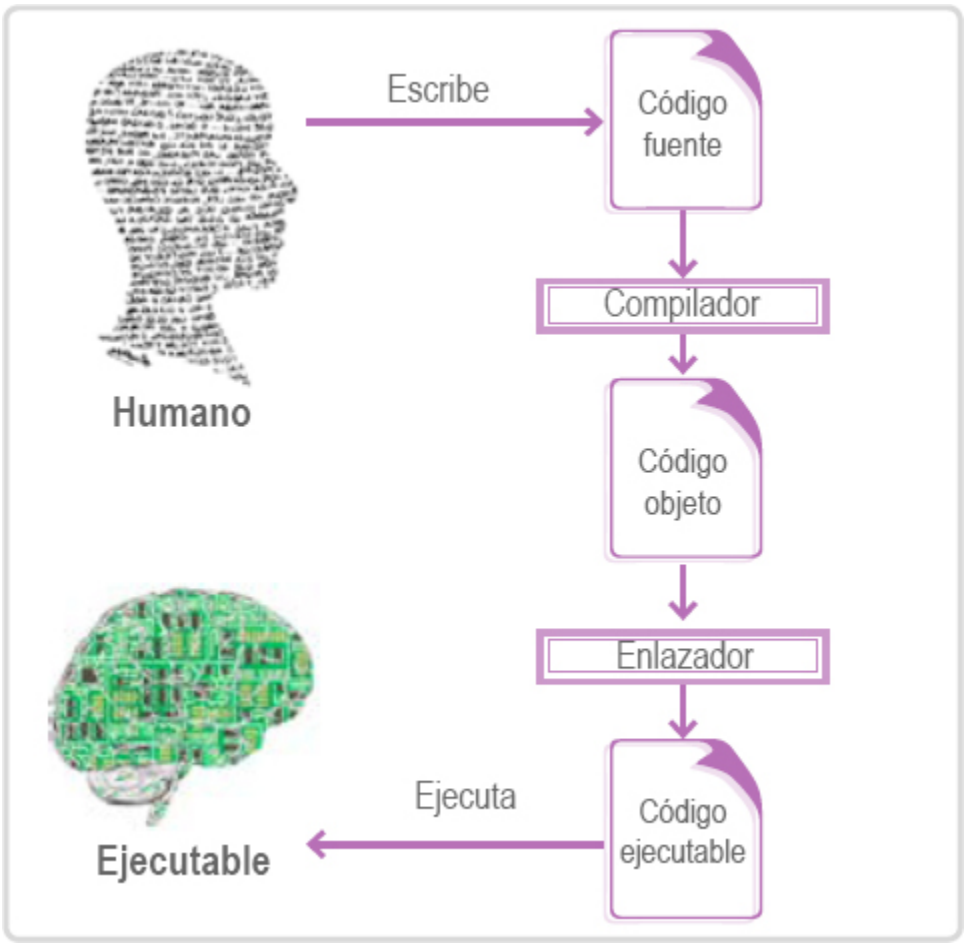


Figura 1. 15. Procedimiento para traducir el código fuente en código ejecutable.

El proceso detallado en la Figura 1.15 puede llevarse a cabo a través de una gran variedad de lenguajes de programación, en el Anexo 1 se aborda, con mayor detalle, el tema de los entornos de desarrollo y la importancia que el lenguaje C++ tiene en el mundo de la computación.

En el video de la barra lateral se presenta lo que el mismísimo Bjarne Stroustrup, creador del lenguaje C++, tiene que decir con respecto a este.

1.6. Piense en grande: Alan Turing

Todos los días se obtienen grandes beneficios gracias a las posibilidades que brinda el mundo de la computación: cada vez que se hace una compra o venta, cuando se accede a internet, al enviar un mensaje multimedia o simplemente al transitar por las calles de una ciudad. Las computadoras y el software están en tantos lugares que pareciera como si siempre hubieran estado ahí. Pero, ¿quién tuvo la idea que llevó a que todo esto fuera posible?, ¿de qué mente visionaria surgió el concepto que hoy brinda tantos beneficios? Revise la liga al video que se presenta en la barra lateral.

Alan Turing imaginó lo que en ese momento parecía imposible, luchó y defendió sus ideas a pesar de las críticas de los supuestos expertos de la época. Sirva esto como un claro ejemplo de lo que el ingenio humano, combinado con el trabajo constante, puede lograr.

Nadie lo sabe, pero de su mente, estimado lector, podría surgir la gran idea que represente el siguiente paso en la evolución tecnológica de la humanidad.

Actividad de repaso

Analice los siguientes problemas y realice, para cada uno, las tres etapas de la fase de resolución del problema. En cada caso se indica la representación del algoritmo que debe presentarse.

Problema 1. Serie aritmética (algoritmo en lenguaje natural)

Elaborar un programa que calcule los dos siguientes números de una serie. Para ello el programa deberá pedir los tres primeros números y calcular automáticamente los dos siguientes para mostrarlos en pantalla. Nota: La secuencia de la serie sólo puede estar determinada por sumas y restas, por ahora no tomes en cuenta series más complejas.

Problema 2. Líneas de producción (algoritmo en lenguaje natural)

En una maquila, un supervisor de producción registró durante un buen tiempo la cantidad de producto terminado que genera diariamente cada línea de producción. Después de dos meses, calculó lo que produce cada una de ellas en promedio por día. Se tienen tres líneas de producción y se desea tener un programa que permita saber cuántos días se necesitan para surtir un pedido de N camisas; la intención es mejorar la planeación de los tiempos de entrega y la de los insumos necesarios para producirlas, ya que últimamente se han registrado retrasos en los tiempos de entrega.

Problema 3. Alerta sísmica (algoritmo en diagrama de flujo)

En el tablero controlador R300 del sistema nacional de alerta sísmica, existe un medidor que indica el número de oscilaciones por minuto que registra un sensor. Se requiere

elaborar un programa que permita mostrar una alerta en la pantalla de los operadores de acuerdo con las siguientes condiciones: si el número de oscilaciones es menor a 20 que la pantalla presente el mensaje “normal”. Si es mayor o igual a 20 y menor a 100 que presente “alerta amarilla” y el factor RK (que resulta de dividir el número de oscilaciones entre 100). Finalmente, si es mayor o igual a 100 que muestre “alerta roja”.

Problema 4. Puerta automática (algoritmo en diagrama de flujo)

Se tienen tres lecturas provenientes de tres sensores de proximidad ubicados en la parte superior de una puerta eléctrica. Para que la puerta se abra es necesario que al menos dos de los tres sensores presenten lecturas menores a 1.5 metros.
Diseña un algoritmo que pida las lecturas de los tres sensores e indique si la puerta debe abrirse o debe permanecer cerrada.

Problema 5. Cotizaciones (algoritmo en diagrama de flujo)

En el gobierno del Estado se está licitando la construcción de una carretera. Se tienen 5 empresas candidatas. Elabora un programa que pida el monto de las 5 cotizaciones. Enseguida se deberá descartar la más barata y la más cara, luego se deberá obtener el promedio de las que restan. El programa deberá mostrar en pantalla las cotizaciones que se eliminan y el promedio del resto de las cotizaciones. Debes pensar en una manera eficiente de descartar usando el menor número de condiciones posible (menos de 10).

Problema 6. Relaciones (algoritmo en pseudocódigo)

Dados tres números determina si la suma de cualquier pareja de ellos es igual al tercer número. Si se cumple esta condición escribir “Existe relación” y mostrar la relación, de lo contrario escribir “No se relacionan”. Por ejemplo 3 9 6 sí tienen relación puesto que 6+3=9.

Problema 7. Piezas lego (algoritmo en pseudocódigo o diagrama de flujo)

Se tiene un conjunto de piezas de lego para construir. Sólo existen dos tipos de piezas, cubos y ruedas. Para formar un auto se requieren 10 cubos y 4 ruedas. Para formar un avión se requieren 15 cubos y 2 ruedas.

Dados el número de cubos y ruedas, determina cuál modelo conviene construir si se quiere construir el mayor número de modelos posible, y otra condición es que todos ellos deben ser del mismo modelo.

Problema 8. Regiones y áreas (algoritmo en lenguaje natural)

Dado un cuadrado con un círculo circunscrito dibujado en el interior, se necesita saber cuál es el área del espacio que está fuera del círculo, pero dentro del cuadrado. Define cuál sería el menor número de datos de entrada que necesitarías y elabora el algoritmo que resuelva el problema.

Problema 9. Máquina envasadora (algoritmo en diagrama de flujo)

Resulta que ha llegado una nueva máquina embotelladora de refrescos, el contenedor principal de la máquina tiene forma cilíndrica. Se sabe que cada envase de refresco debe contener M mililitros. Se desea saber cuántos refrescos puede llenar la máquina de una sola vez, sin recargar el contenedor. Sólo se tienen los datos del radio de la base y la altura, ambos medidos en metros.

Problema 10. Promedio (algoritmo diagrama de flujo)

Dadas tres calificaciones se requiere saber si el promedio es superior o igual a 70, de ser así deberá mostrar el mensaje “Aprobado” y mostrar el promedio calculado; de lo contrario deberá mostrar sólo el mensaje “Reprobado” y el número de puntos que le faltaron para aprobar.

Problema 11. Transferencias (algoritmo en pseudocódigo)

Elabore un programa que pida el monto de 6 transacciones bancarias, enseguida la computadora deberá mostrar la suma

total de las transacciones y el número de transacciones de monto mayor o igual a 15,000 pesos.

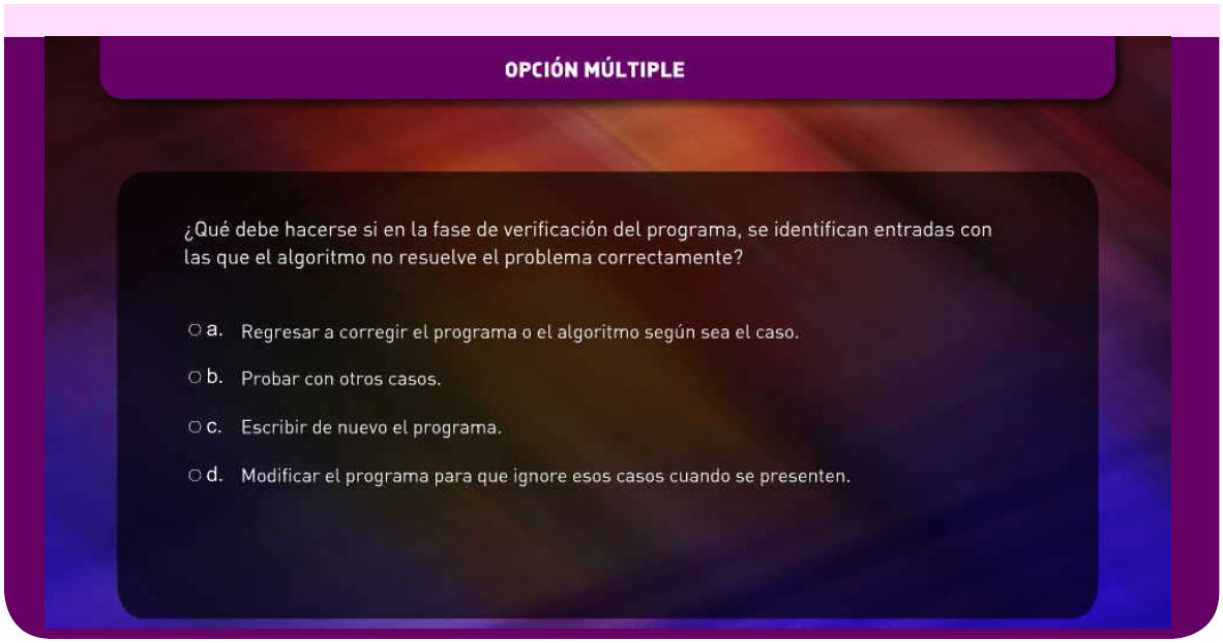
Problema 12. Clientes preferenciales (algoritmo en diagrama de flujo)

El negocio de don Rómulo es una empresa cuya actividad comercial es la de vender libretas por mayoreo. El precio unitario de la libreta varía dependiendo de la cantidad de libretas que el cliente solicite.

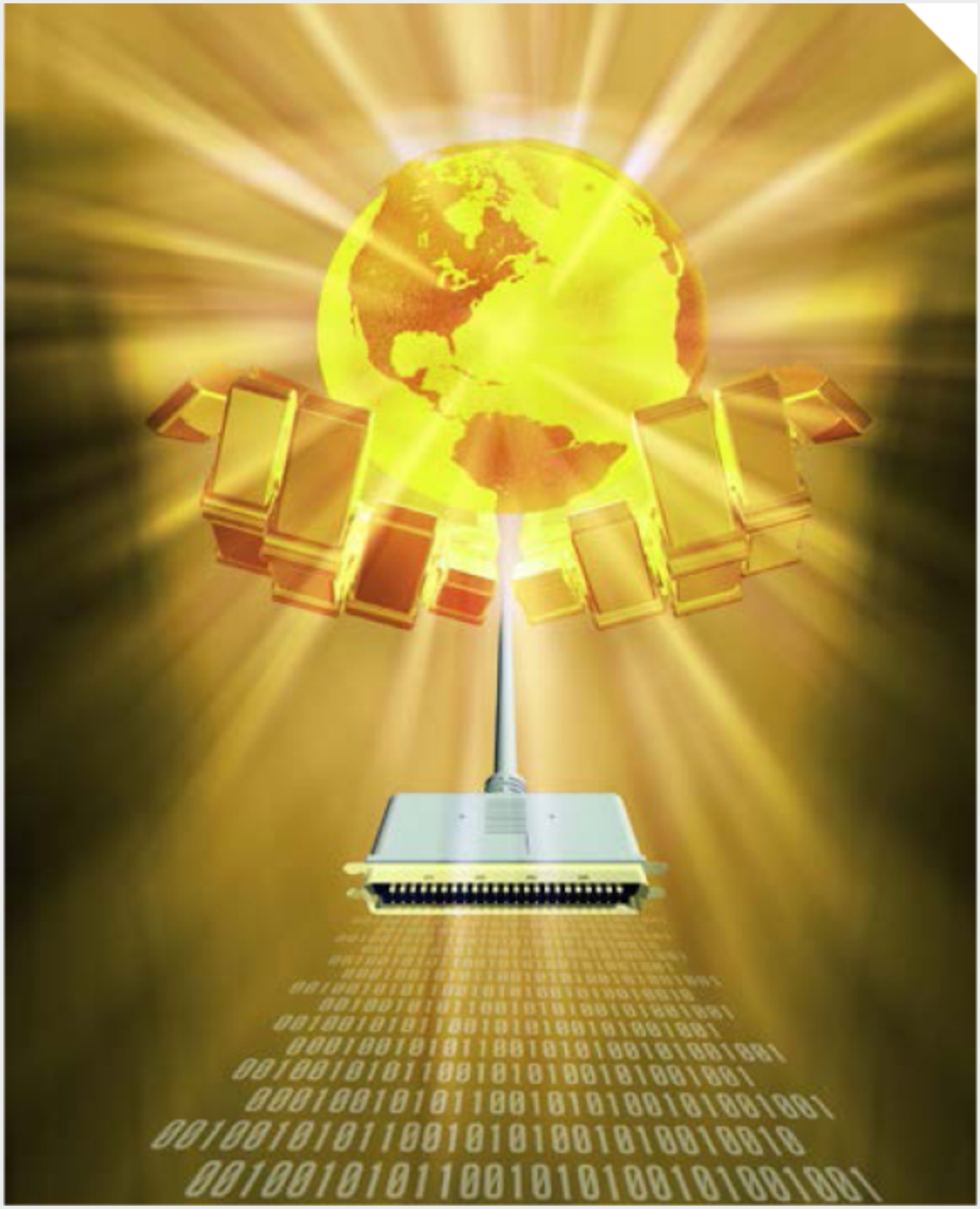
Desde 1 a 100 libretas el precio unitario es de $6.00, de 101 a 1000 el precio es de $5.20 y de 1001 en adelante el precio es de $4.50 por pieza.

Se requiere un algoritmo que obtenga la cantidad de libretas que va a comprar. Dependiendo de ello deberá calcularse y mostrarse el total a pagar.

Ejercicio integrador del capítulo 1



Conclusión del capítulo 1



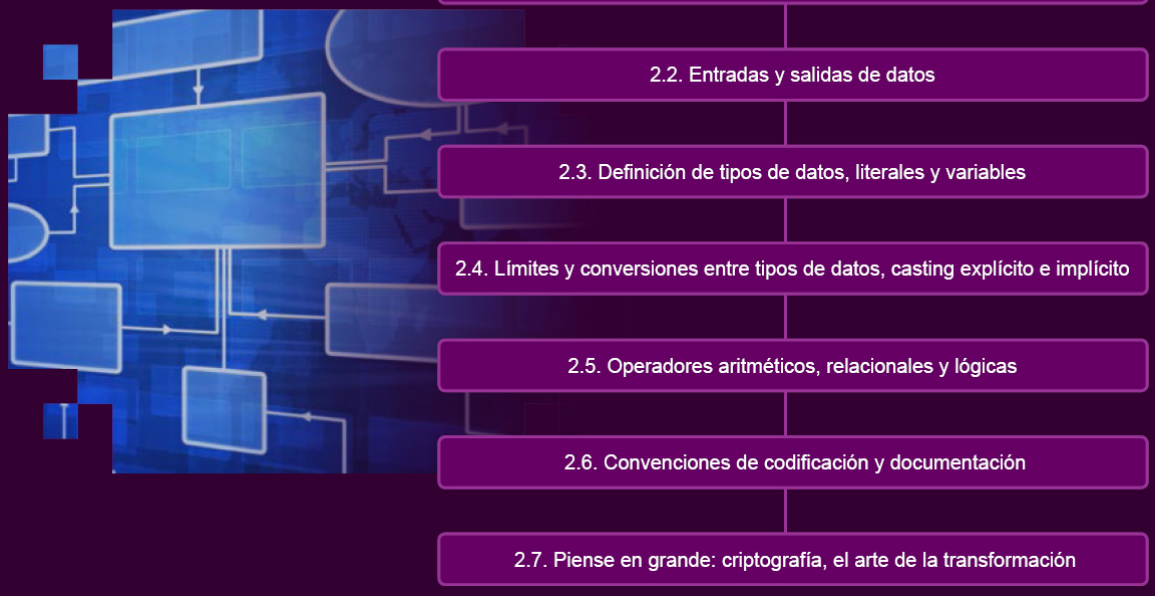
Como se vio en este capítulo, el proceso para solucionar problemas a través de la programación no es una tarea sencilla, requiere de una serie de conocimientos y habilidades específicas. La buena noticia es que, sin saberlo, todos han trabajado en el desarrollo de dichas habilidades, en mayor o menor medida, a lo largo de la propia formación académica e incluso desde la infancia temprana.

Las personas diseñan algoritmos todos los días sin pensarlo siquiera: por las mañanas al determinar el orden de las actividades para llegar temprano a trabajar en función del tiempo restante; al jugar cuando se piensa en una estrategia para vencer al oponente en función de su pericia; mientras se trabaja y se busca la manera de mejorar un proceso para ahorrar tiempo o dinero.

Los siguientes capítulos, abordarán los diferentes aspectos del lenguaje C++; a fin de que cada lector se convierta, gradualmente, en el autor de sus propias herramientas tecnológicas. Las computadoras han llegado para quedarse, y el profesionista que sea capaz de programarlas, irá siempre un paso delante de los demás.

Capítulo 2. Conceptos básicos

Organizador temático



Conceptos básicos

2.1. Estructura básica de un programa

Al igual que cuando se aprende un nuevo idioma, al aprender un lenguaje de programación el primer reto consiste en identificar las reglas básicas que lo rigen. Partiendo de un algoritmo diseñado para resolver un problema, el reto está en encontrar la manera de traducirlo al lenguaje de programación C++. Vale la pena recalcar que de acuerdo con el proceso general presentado en la Figura 1.3, no se recomienda iniciar con la escritura de un programa hasta no tener claro el algoritmo que se desea representar; hacerlo

sería equivalente a comenzar a pronunciar palabras, sin saber qué es lo que se quiere decir.

En la Figura 2.1 se observa la relación entre el algoritmo que calcula el rendimiento de un automóvil y el código fuente correspondiente, escrito en el lenguaje C++.

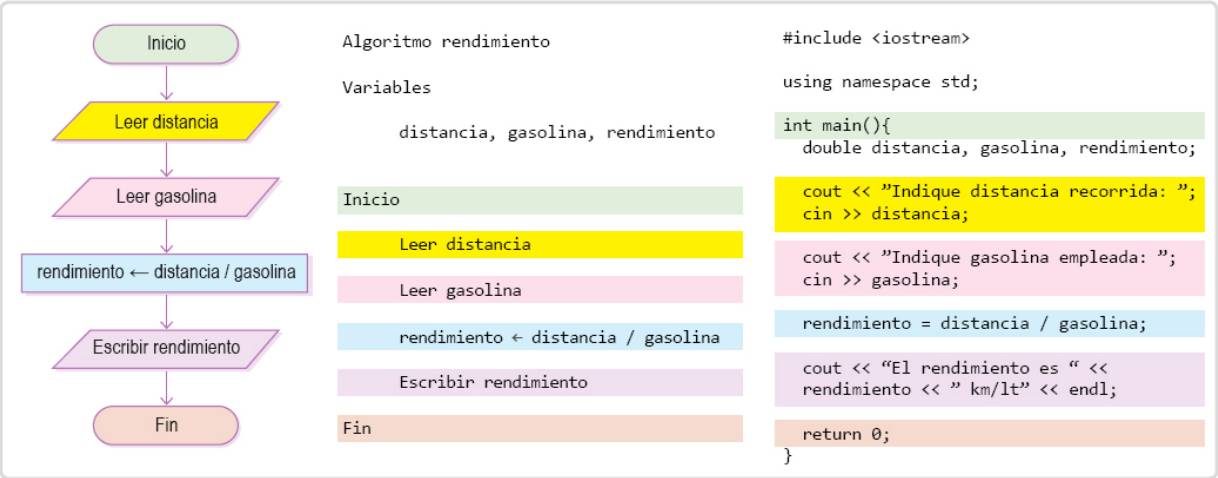
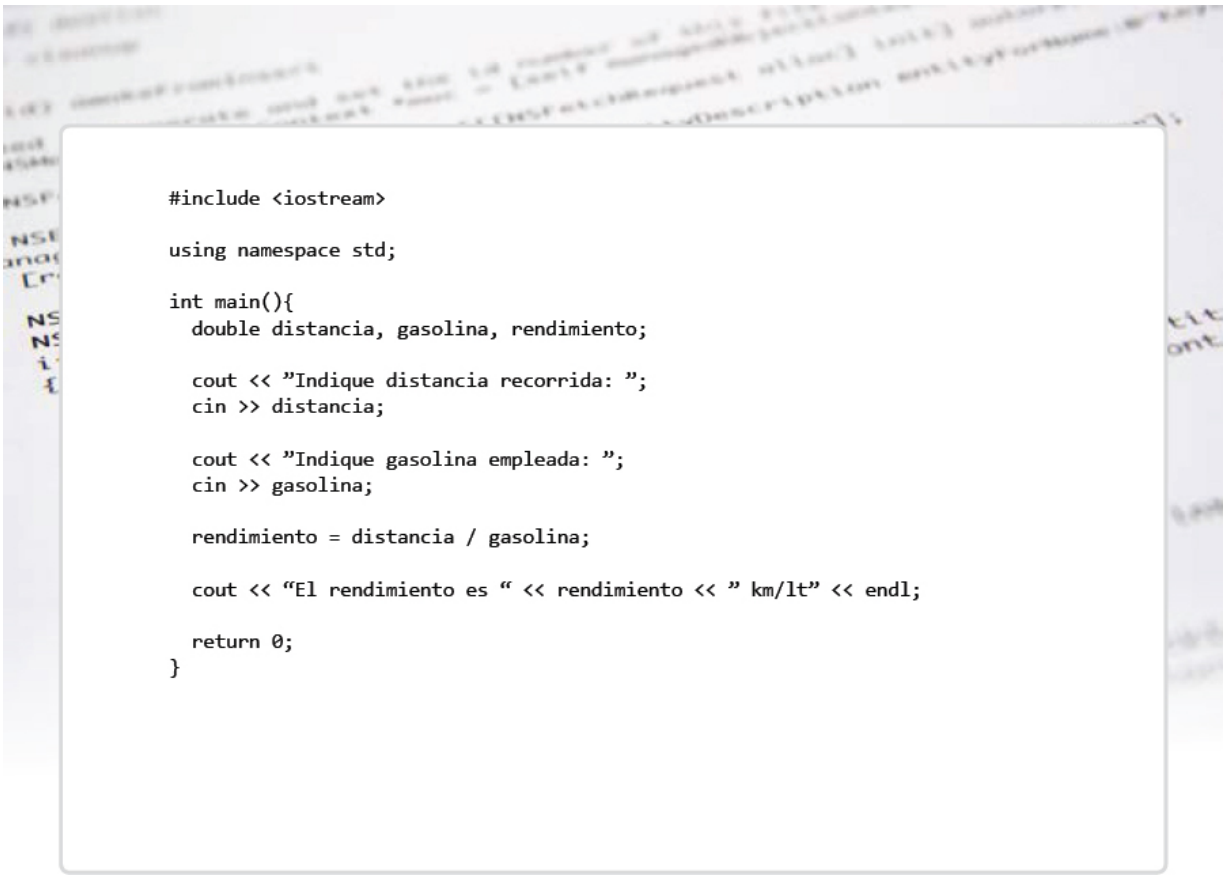


Figura 2. 1. Relación entre el algoritmo y el código fuente correspondiente en C++.

Es muy importante que el lector note que el programa contiene un número de instrucciones mayor al número de pasos que presenta el algoritmo correspondiente. Esto se debe a que en el programa no sólo se escriben instrucciones asociadas con acciones, también se escriben instrucciones declarativas. Se le denomina declaración a un segmento de código que tiene como objetivo especificar una propiedad, comportamiento o característica, ya sea del programa en su totalidad o de una parte del mismo.

Las instrucciones que realizan acciones tienen un efecto claro dentro del programa: obtener un dato, realizar un cálculo, mostrar información en pantalla y otras acciones similares, de ahí que sea fácil identificarlas. Las instrucciones declarativas por el contrario, no están asociadas con el algoritmo y no tienen un efecto perceptible durante la ejecución del programa, de ahí que con frecuencia causen confusión a quienes se inician en el lenguaje de la programación.

Sin importar si son instrucciones declarativas o no, todas tienen una sintaxis definida de acuerdo con el lenguaje C++. Explore, en la Figura 2.2, las diferentes instrucciones que forman parte del programa y conocer su uso.



```
#include <iostream>

using namespace std;

int main(){
  double distancia, gasolina, rendimiento;

  cout << "Indique distancia recorrida: ";
  cin >> distancia;

  cout << "Indique gasolina empleada: ";
  cin >> gasolina;

  rendimiento = distancia / gasolina;

  cout << "El rendimiento es " << rendimiento << " km/lt" << endl;

  return 0;
}
```

Figura 2. 2. Programa que calcula en rendimiento de gasolina y las instrucciones que lo conforman.

Para entender el funcionamiento del programa es importante comprender con claridad los conceptos que intervienen en su definición. A continuación se explican algunos de los conceptos mencionados en la Figura 2.2.

Biblioteca: Un conjunto de elementos predefinidos del lenguaje que pueden ser incluidos y utilizados en la creación de nuevos programas.

Espacio de nombres: Una agrupación lógica de un conjunto de elementos. Dos elementos se pueden llamar igual, siempre y cuando pertenezcan a espacios de nombres distintos. Si en un momento dado existe ambigüedad, se coloca el nombre del espacio de nombres como prefijo del elemento.

Función: Es un conjunto de instrucciones agrupadas bajo un nombre. Una función puede recibir parámetros como información de entrada y devuelve un valor de salida. La función main en particular es una función especial, ya que es el punto de entrada del programa.

Variable: Una variable es un identificador utilizado para representar un dato de manera simbólica, puede cambiar durante la ejecución del programa. Un identificador está conformado por una secuencia de caracteres, debe iniciar con una letra, seguida de letras, dígitos o guiones bajos. Un identificador válido no debe contener espacios.

Bloque: Es un conjunto de instrucciones asociadas a una estructura de programación, regularmente los bloques están delimitados por una llave de apertura y una llave de cierre.

Ámbito: Se refiere a la región del programa donde una variable es accesible. El ámbito de una variable se reduce al bloque dentro del cual fue declarada, a partir del punto donde fue declarada hacia adelante. En el capítulo 3 se observará con mayor claridad el concepto de ámbito.

Como puede apreciarse, conocer la terminología empleada en el ámbito de la programación es muy importante para comprender los textos y los artículos del área. De igual manera, es necesario conocer con precisión las implicaciones que las instrucciones tienen en el funcionamiento de los programas. Muchas personas saben

que los programas funcionan en la computadora, y que ésta a su vez está compuesta por una gran variedad de componentes físicos y lógicos: procesador, memoria, disco duro, **archivos** y otros más. Para quien simplemente usa el software, la relación de éste con los componentes de la computadora es transparente, sin embargo, a la hora de escribir programas es necesario conocer un poco más sobre estas interacciones.

Navegue por la Figura 2.3 para conocer el funcionamiento, paso a paso, del programa del rendimiento de gasolina y su relación con los diferentes componentes de la computadora.

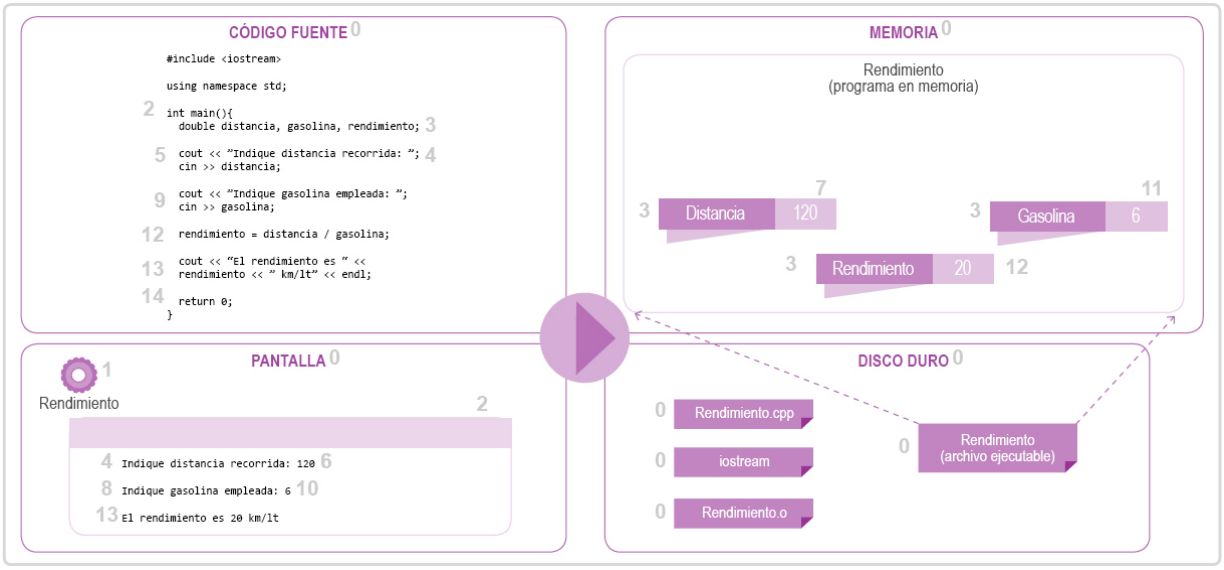


Figura 2. 3. Funcionamiento del programa de rendimiento de gasolina paso a paso.

Como se mencionó con anterioridad, no todas las instrucciones generan un efecto directo en pantalla, sin embargo, todas forman parte de la secuencia de instrucciones necesarias para llegar al resultado esperado. En las siguientes secciones se aborda cada uno de los aspectos de la programación con la finalidad de hacer el tema más accesible el lector.

2.2. Entradas y salidas de datos

Como se vio antes, los programas requieren interactuar con el usuario de diferentes maneras. La más común es por medio del teclado, que está representado en C++ por *cin*. De la misma forma, si la computadora necesita mostrar un resultado al usuario, lo más común es enviarlo a pantalla, para ello se usa en C++ *cout.*

Tanto cin como cout son ***streams***, un stream es un flujo de datos que sirve para transferir información de un elemento a otro. En el caso del ejemplo, cout es usado para mostrar un mensaje textual al usuario, así como el valor contenido en una variable, al transferirlos de la memoria de la computadora a la pantalla. De manera contraria, cin se usa para almacenar un valor en una variable, lo transfiere del teclado al espacio de memoria correspondiente a dicha variable. Este no es el único uso que pueden tener los streams, como se verá más adelante, la misma lógica se aplica para el manejo de archivos.

Observe cuidadosamente que los símbolos << y >> se utilizan dependiendo de la dirección en que se transfiere la información. En la Figura 2.4 puede apreciarse el fenómeno descrito para el caso de cout.

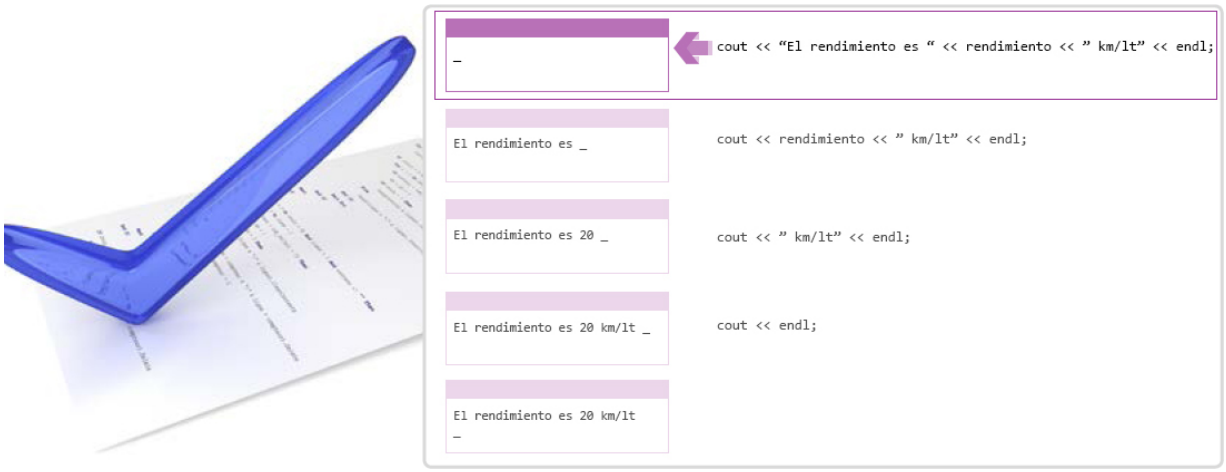


Figura 2. 4. Flujo de datos en dirección hacia pantalla.

Es muy importante hacer notar que también existe en C++ un stream llamado *cerr,* que representa la salida de errores estándar. Automáticamente este flujo de datos será direccionado también a la

pantalla. Cada **sistema operativo** tiene mecanismos para establecer el destino de cada stream, de tal forma que la salida de errores estándar podría redirigirse a un archivo o a un *socket.* También existe el stream clog que también está asociado automáticamente con la pantalla, cuyo objetivo principal enviar información a una bitácora en donde se almacena información sobre la ejecución del programa. Lo más común es redirigir este stream a un archivo.

Formato de salida de datos

Para dar formato al flujo de datos que se envían a un stream, existen una serie de funciones predefinidas en la biblioteca *iomanip,* cada una de éstas lleva a cabo una actividad específica. En la Tabla 2.1 se muestran algunas funciones, su sintaxis y un ejemplo de su uso:

Tabla 2.1 Manipuladores del formato de salida

Función	Uso	Ejemplo de uso
setw(n)	Establece el número mínimo de caracteres para representar el siguiente dato. Si el dato ocupa menos espacios, se añaden espacios en blanco.	`cout << setw(10) << "uno"; uno`
setfill(c)	Establece el caracter que será utilizado para rellenar espacios en blanco.	`cout << setfill('-') << setw(5) << "uno"; --uno`
left	Establece alineación a la izquierda.	`cout << setfill('-') << left << setw(5) << "uno"; uno--`
right	Establece alineación a la derecha.	`cout << setfill('?') << right << setw(5) << "uno"; ??uno`
fixed	Establece notación de punto decimal para mostrar un número de punto flotante.	`cout << fixed << 2540001.23; 2540001.23`

Función	Uso	Ejemplo de uso
scientific	Establece notación científica para mostrar un número de punto flotante.	`cout << fixed << 254000123; 2.54000123e+006`
setprecision(n)	Establece el número de dígitos después del punto decimal para números de punto flotante.	`cout << fixed << setprecision(4) << 2540001.23 2540001.2300`

Los manipuladores representan una funcionalidad muy útil sirven para presentar información en pantalla con cierta alineación, como por ejemplo, los montos económicos.

2.3. Definición de tipos de datos, variables y constantes

2.3.1 Tipos de datos y variables

Para que los programas almacenen datos, éstos deben representarse simbólicamente por medio de identificadores. A estos identificadores, que representan datos en programación, de les denomina variables. Cada variable debe tener un tipo de dato que debe almacenar, dependiendo de la naturaleza del dato.

Hay cinco tipos de datos básicos:

char: representa al tipo **caracter** (letra, dígito o símbolo).

int: representa al tipo entero.

bool: representa un valor de verdad (verdadero o falso).

float: representa un número de punto flotante.

double: representa un número de punto flotante de mayor precisión que float.

Además, C++ cuenta con cuatro modificadores que pueden alterar el rango de valores que almacena un tipo de datos específico:

short: Se aplica para acortar la cantidad de memoria que un tipo utiliza y, en consecuencia, el rango de valores posibles que puede almacenar la variable.

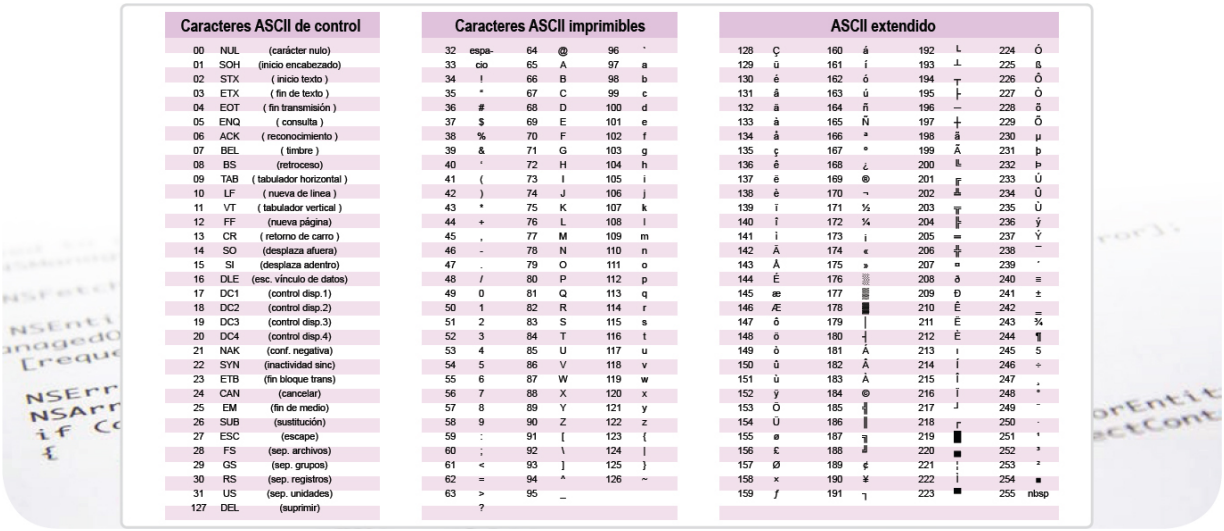
long: Se aplica para acrecentar la cantidad de memoria que un tipo utiliza y, en consecuencia, el rango de valores posibles que puede almacenar la variable.

signed: Especifica que la variable puede contener tanto valores positivos como negativos. Este es el modo predefinido de todas las variables declaradas.

unsigned: Especifica que la variable sólo puede contener números positivos, lo que elimina la posibilidad de tener un valor con signo negativo. Esto incrementa el valor máximo positivo que la variable puede representar, ya que el ***bit*** de signo se utiliza para almacenar más información.

El número de ***bytes*** que utiliza cada tipo no está especificado por C++, depende de la implementación particular del compilador y de la plataforma. La única restricción es que exista coherencia entre tipos, es decir, una variable de tipo *double* debe usar igual o mayor al número de bytes que una de tipo *float;* una de tipo *long int* debe usar un número de bytes mayor o igual que una de tipo *int* y así sucesivamente.

Es importante recalcar que en los casos de *char* y *bool,* existen representaciones numéricas para ambos. En el caso del tipo char, el valor numérico se determina con el código ASCII que corresponde al caracter de acuerdo con lo que se muestra en la Figura 2.5.



Caracteres ASCII de control

00	NUL	(carácter nulo)
01	SOH	(inicio encabezado)
02	STX	(inicio texto)
03	ETX	(fin de texto)
04	EOT	(fin transmisión)
05	ENQ	(consulta)
06	ACK	(reconocimiento)
07	BEL	(timbre)
08	BS	(retroceso)
09	TAB	(tabulador horizontal)
10	LF	(nueva de línea)
11	VT	(tabulador vertical)
12	FF	(nueva página)
13	CR	(retorno de carro)
14	SO	(desplaza afuera)
15	SI	(desplaza adentro)
16	DLE	(esc. vínculo de datos)
17	DC1	(control disp.1)
18	DC2	(control disp.2)
19	DC3	(control disp.3)
20	DC4	(control disp.4)
21	NAK	(conf. negativa)
22	SYN	(inactividad sinc)
23	ETB	(fin bloque trans)
24	CAN	(cancelar)
25	EM	(fin de medio)
26	SUB	(sustitución)
27	ESC	(escape)
28	FS	(sep. archivos)
29	GS	(sep. grupos)
30	RS	(sep. registros)
31	US	(sep. unidades)
127	DEL	(suprimir)

Caracteres ASCII imprimibles

32	espa-	64	@	96	`
33	cio	65	A	97	a
34	!	66	B	98	b
35	"	67	C	99	c
36	#	68	D	100	d
37	$	69	E	101	e
38	%	70	F	102	f
39	&	71	G	103	g
40	'	72	H	104	h
41	(	73	I	105	i
42	)	74	J	106	j
43	*	75	K	107	k
44	+	76	L	108	l
45	,	77	M	109	m
46	-	78	N	110	n
47	.	79	O	111	o
48	/	80	P	112	p
49	0	81	Q	113	q
50	1	82	R	114	r
51	2	83	S	115	s
52	3	84	T	116	t
53	4	85	U	117	u
54	5	86	V	118	v
55	6	87	W	119	w
56	7	88	X	120	x
57	8	89	Y	121	y
58	9	90	Z	122	z
59	:	91	[	123	{
60	;	92	\	124	|
61	<	93	]	125	}
62	=	94	^	126	~
63	>	95	_		
	?				

ASCII extendido

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Figura 2. 5. Códigos ASCII.

Por su parte el tipo bool también tiene una representación numérica, su valor *false* es traducido como 0 y *true* es traducido como 1.

En la Figura 2.6 se observan las implicaciones en cuanto a tamaño y rango de cada modificador sobre los diferentes tipos de datos.

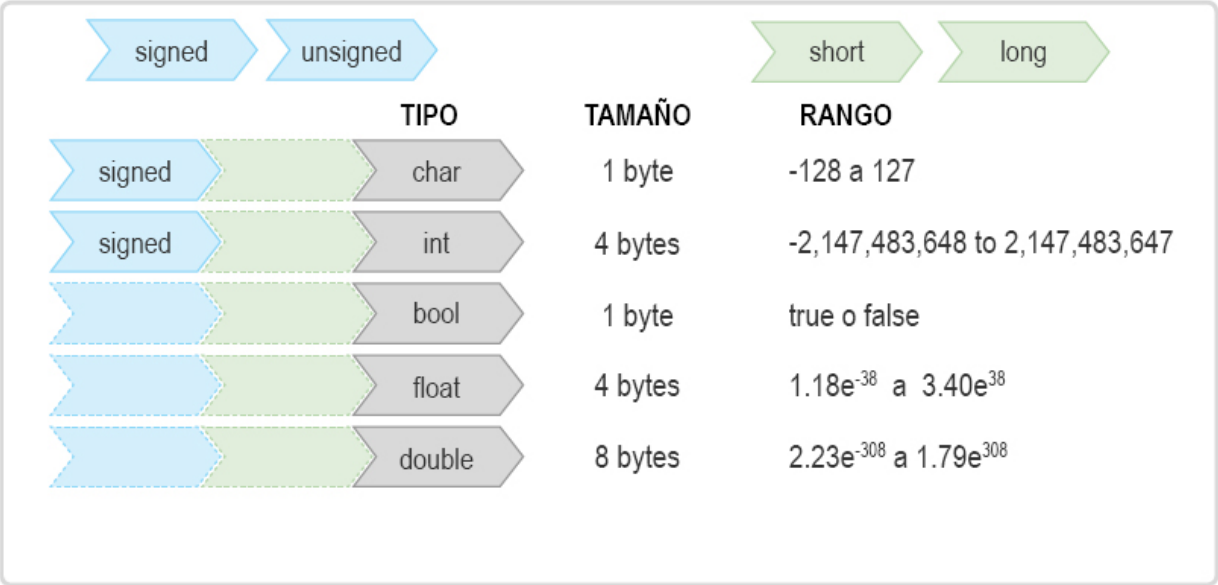


Figura 2. 6. Variación del rango de un tipo de acuerdo con los modificadores.

Cabe resaltar que por cuestiones de eficiencia, se recomienda emplear el tipo de datos adecuado en función del rango de datos que almacenará. No se recomienda usar innecesariamente tipos de datos de mayor tamaño, ya que esto implica hasta el doble o el triple de procesamiento para la computadora.

2.3.2 Representación y almacenamiento de datos

Para tener una idea más clara de la razón por la que los rangos son tan específicos es necesario comprender bien el concepto de bit. A través de un bit puede representarse cualquier valor imaginable.

Observe video de la barra lateral donde el Dr. Adrián Arnoldo Paenza explica claramente el fascinante mundo de los números binarios.

2.3.3 Constantes

Hasta este punto se ha visto que cualquier variable, sin importar su tipo, puede ser modificada dentro de un programa tantas veces como el problema lo requiera. Sin embargo, en algunos casos es necesario definir valores constantes que no se modifican durante

todo el programa. Con frecuencia estos valores son factores o valores conocidos de cierto dominio.

Para la definición de estos valores se utiliza la palabra reservada *const,* esta palabra debe anteponerse a la declaración del tipo y nombre. Además, su uso implica que el identificador debe ser inicializado. Algunos ejemplos de constantes son:

```
const int N = 30;

const double FACTOR = 45.67;

const short int PESO = 2;
```

Es importante recalcar que para las constantes aplica también la regla del ámbito de una variable, es decir, sólo son válidas dentro del bloque donde fueron declaradas y partir del punto en que se definieron.

2.4. Límites y conversiones entre tipos de datos, casting explícito e implícito

Al trabajar con diferentes tipos de datos es inevitable que, eventualmente, se mezclen entre sí. Como consecuencia del mismo planteamiento del problema, es común encontrar datos de diferentes tipos en una expresión aritmética. C++ tiene reglas bien definidas que permiten eliminar la ambigüedad que pueda presentarse:

REGLA 1

Si dos operandos asociados a un operador binario son ambos algún tipo de entero, se convierten ambos al tipo int.

REGLA 2

Si dos operandos asociados a un operador binario tienen diferente tipo, deben convertirse a un tipo común. El tipo común debe ser el de mayor precisión.

Las mismas reglas se siguen en el caso de las asignaciones. Con la diferencia de que la variable que recibe el resultado no puede cambiar su tipo de dato, dado que ya fue declarado previamente.

Cuando la conversión implica una pérdida de precisión, algunos compiladores emiten un mensaje de advertencia, otros, simplemente hacen la operación y truncan o convierten la representación correspondiente. Utilice la Figura 2.7 para observar la aplicación de las reglas antes descritas para los diferentes casos que se presentan.

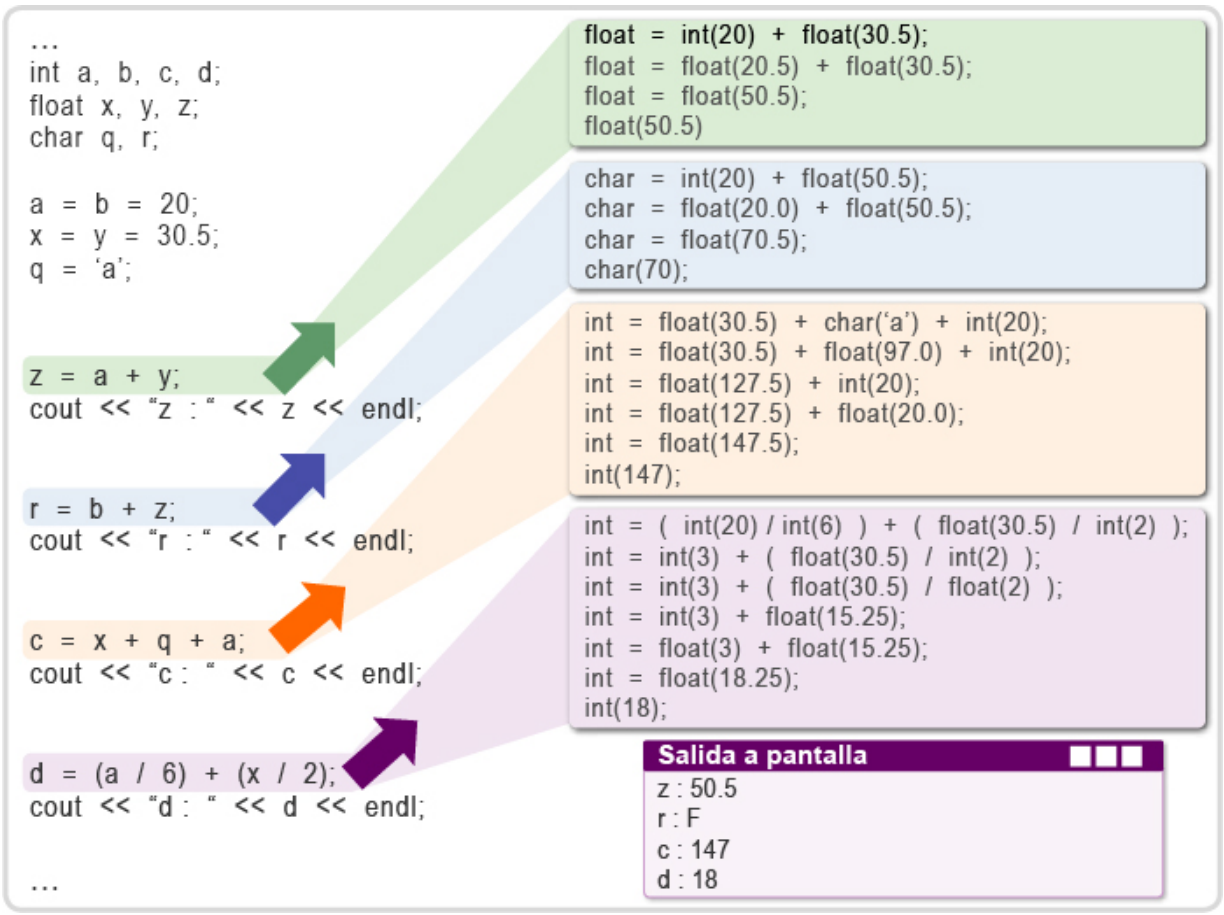


Figura 2. 7. Evaluación de expresiones conversiones implícitas.

En los casos en los que se presenta una pérdida de precisión es recomendable hacer una conversión explícita, o *casting.* Esto le hace saber al compilador, que el programador está consciente de la pérdida de precisión. Además evita comportamientos anómalos difíciles de detectar. El programador puede usar el casting explícito siempre que lo requiera, por ejemplo, para obtener el entero inmediato superior de un valor de punto flotante.

```
   int sigEntero;
   float valor = 34.66;
   sigEntero = (int) valor + 1;
   cout << "La parte entera de " << valor << " es " <<
sigEntero << endl;
```

Como se observa, el lenguaje cuenta con una gran variedad de tipos y modificadores así como de reglas definidas para su adecuada interpretación. De entrada esto podría parecer excesivo, pero esta flexibilidad se requiere debido a que C++ es un lenguaje diseñado para resolver las necesidades de múltiples sistemas y plataformas: Desde un programa para simular el comportamiento de las galaxias en una **supercomputadora**, hasta un programa para estacionar automáticamente un automóvil en la computadora central del mismo. Como se observa, la diferencia en cuanto a poder de procesamiento y memoria es abismal entre un caso y otro. C++ está diseñado para generar programas en ambos entornos y para ser eficiente a distintos niveles, por lo que es natural encontrarse con múltiples alternativas para el manejo de datos.

2.5. Operadores aritméticos, relacionales y lógicos

Como se ha visto, la amplia variedad de problemas y la diversidad de dominios en los que la programación de utiliza, requieren una gran maleabilidad de las operaciones que puedan realizarse con los datos. De ahí la diversidad de operadores que maneja el lenguaje de programación C++ que van desde operadores aritméticos hasta operadores a nivel de bits, que es la unidad más pequeña y elemental de información que se tiene. Ahora se procederá a revisar los diferentes tipos de operadores disponibles:

Independientemente del tipo de dato que manejen los operadores pueden ser unarios o binarios dependiendo del número de operandos sobre el que operan.

Operadores aritméticos: Realizan operaciones numéricas con los datos. Son los más comunes, se ven continuamente en matemáticas. Las principales diferencias radican en que se usa el caracter asterisco para multiplicar y el caracter diagonal para dividir. Además está el operador módulo, o residuo, que permite obtener el

residuo de la división entera. En la Tabla 2.2 se muestran estos operadores:

Tabla 2.2 Operadores aritméticos

Nombre	Tipo	Sintaxis
Signo positivo	unario	+ operando
Signo negativo	unario	- operando
Suma	binario	operando + operando
Resta	binario	operando - operando
Multiplicación	binario	operando * operando
División	binario	operando / operando
Módulo	binario	operando % operando

Cabe aclarar que el operador módulo es un operador muy utilizado en programación pues permite realizar aritmética modular. El resultado de la expresión *a % b* es el residuo de la división entera de *a / b.* Este es el único operador aritmético que sólo puede ser usado con números enteros.

Operadores relacionales: Estos operadores tienen la finalidad de verificar relaciones entre operandos, el resultado de las operaciones que se realizan con éstos es un valor de verdad, es decir, sólo hay dos resultados posibles: verdadero o falso. En la Tabla 2.3 se muestran los operadores relacionales:

Tabla 2.3 Operadores relacionales

Nombre	Tipo	Sintaxis
Signo positivo	unario	+ operando
Signo negativo	unario	- operando
Suma	binario	operando + operando
Resta	binario	operando - operando
Multiplicación	binario	operando * operando
División	binario	operando / operando
Módulo	binario	operando % operando

Es muy importante tomar en consideración que cada uno de los denominados operandos puede ser una expresión aritmética completa. Después de evaluar cada una de ellas el operador relacional determinará la veracidad o falsedad de la afirmación. En el caso de C++ también acepta números enteros como expresiones lógicas válidas. La correspondencia es la siguiente:

- Si el valor entero 0 la computadora lo interpreta como falso.
- Si es cualquier otro número entero la computadora lo interpreta como verdadero.

Operadores lógicos: Inciden sobre las expresiones lógicas operando sobre sus valores de verdad. Esto permite escribir expresiones compuestas. En la Tabla 2.4 se muestran los operadores lógicos:

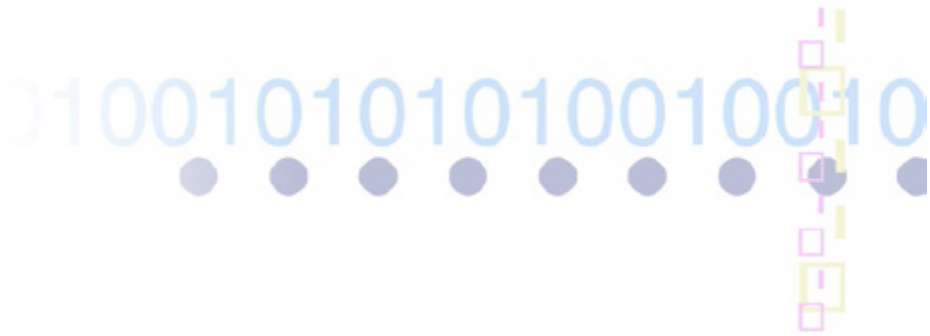
Tabla 2.4 Operadores lógicos

Nombre	Tipo	Sintaxis
Operador AND	binario	operando && operando

Nombre	Tipo	Sintaxis
Operador OR	binario	operando || operando
Operador NOT	unario	! operando

Las reglas básicas para determinar el valor de verdad resultante son muy sencillas:

Para el caso del AND, la expresión resultante es verdadera si y sólo si ambas expresiones son verdaderas. En el caso del OR basta con que una de las dos expresiones lo sea para que la expresión resultante sea verdadera.



De la misma forma que las expresiones aritméticas, las expresiones compuestas se evalúan de izquierda a derecha por pares. Si se desea establecer una precedencia diferente, pueden utilizarse paréntesis para especificarla.

2.5.1 Precedencia de operadores

Ahora que se han planteado los diferentes tipos de operadores, debe considerarse que todos se usan de manera combinada, como puede observarse en el siguiente ejemplo:

```
...
int a ← 4
```

```
int b ← 3

int c ← 5

si (b==a%2 && a<c || a>b–1 && a<c*2) Entonces {

// Instrucciones

}

...
```

En el ejemplo anterior se observa el uso de expresiones aritméticas, relacionales y lógicas combinadas en una sola línea. Todo esto es posible debido a una jerarquía existente llamada precedencia de operadores. Esta jerarquía elimina la ambigüedad al establecer el orden en que se aplicarán los operadores. En la Tabla 2.5 se muestran los operadores vistos hasta ahora y su precedencia:

Tabla 2.5 Precedencia de operadores

Operadores	Nombre	Asociatividad
( )	Paréntesis.	Izquierda a derecha
! + -	Negación, signo positivo y negativo	Derecha a izquierda
* / %	Multiplicación, división y residuo	Izquierda a derecha
+ -	Suma y resta	Izquierda a derecha
<< >>	Direccionamiento	Izquierda a derecha
< <= > >=	Menor que, menor o igual que, mayor que o mayor o igual que	Izquierda a derecha
== !=	Igual que, diferente de	Izquierda a derecha
&&	AND lógico	Izquierda a derecha
||	OR lógico	Izquierda a derecha
=	Asignación	Derecha a izquierda

El orden de precedencia comienza por los paréntesis y termina por el operador de asignación en orden de mayor a menor precedencia. Cabe resaltar que no todos los operadores de evalúan de izquierda a derecha. Los operadores unarios y el operador asignación, por su naturaleza, se evalúan de derecha a izquierda. Revise la Figura 2.4 para identificar el orden de aplicación de los operandos del ejemplo, use como referencia la Tabla 2.8.

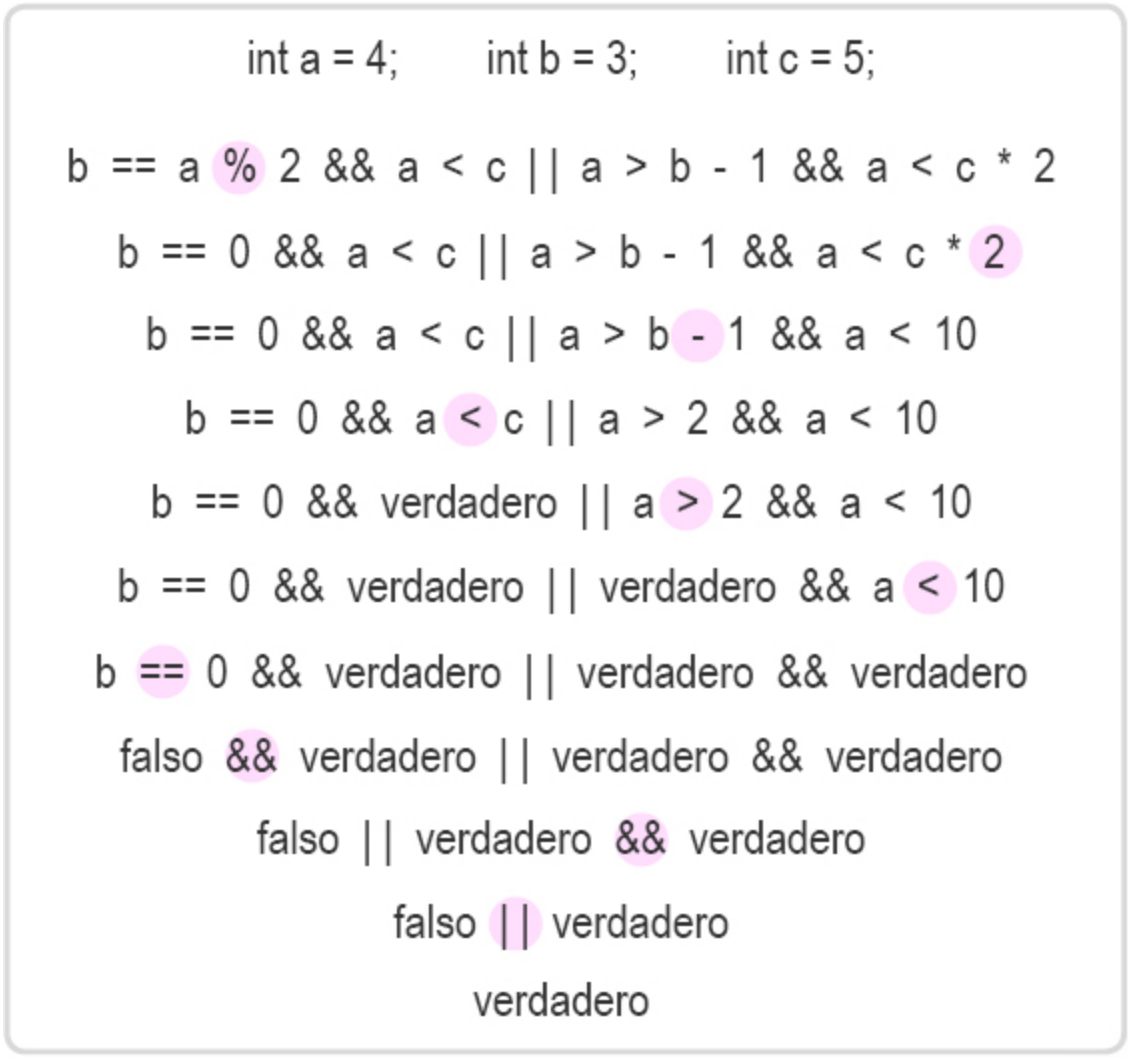


Figura 2. 8. Orden de ejecución de una expresión de acuerdo con la precedencia de operadores.

Revise el video que se encuentra en la barra lateral.

Por supuesto, estos no son todos los operadores que maneja C++, pero son los que se han abordado hasta el momento.

2.6. Convenciones de codificación y documentación

Cuando se interpreta un programa escrito en un lenguaje de programación, la computadora no repara en la existencia de líneas, no toma en cuenta los espacios, mucho menos los comentarios al margen. Para la computadora una instrucción

termina en donde hay un punto y coma, no importa si la instrucción está conformada por varias líneas o por una sola.

Sin embargo, los programas escritos en C++ no sólo se leen por la computadora, sino también por otras personas. Para nadie es un secreto que en grandes proyectos es indispensable el trabajo en equipo. De ahí que una recomendación importante es que el código fuente debe ser claro y legible para las personas. Si el programa es correcto, la computadora lo interpretará sin problema, pero aun siendo correcta la sintaxis, un programa escrito de manera desordenada es ilegible y, por ende, difícil de reusar, mantener o modificar. Por esta razón, se han definido diversos estilos de codificación y documentación, en el Anexo 2 se abordan los estilos de codificación más utilizados.

2.7. Piense en grande: criptografía, el arte de la transformación

Como se ha visto a lo largo del capítulo, la representación de la información es un factor importantísimo a la hora de escribir programas. Sin embargo, en muchas ocasiones, y cada vez con mayor frecuencia, es necesario transformar la información para que no pueda ser vista por otras personas. Obviamente se trata de información confidencial como contraseñas, números de cuenta, archivos e imágenes. Esta transformación, además de ser efectiva, debe ser reversible, es decir, debe ser posible procesar la información que se transformó para obtener nuevamente la información original. El área de las ciencias computacionales que se encarga de estudiar estas transformaciones se llama criptografía. Varias de las técnicas empleadas, utilizan dos de los elementos revisados en el capítulo: la representación de la información, tipos de datos, y los operadores aritméticos, relacionales y lógicos.

Revise el video ¿Qué es la criptografía?, ubicado en la barra lateral para ver lo que hay detrás del arte de la transformación.

El concepto de la criptografía es bastante antiguo y la idea muy sencilla, no obstante, sigue siendo un tema de investigación abierto y su límite está aún muy lejos de alcanzarse. En el video *Cryptography Research,* ubicado en la barra lateral puede revisar lo que dicen los artífices de este tema.

De la misma forma en que empresas como IBM están preocupadas por la seguridad de la información, hay un amplio grupo de personas u organizaciones que están interesadas justo en lo contrario.

Es claro que en la sociedad actual, las computadoras juegan un papel cada vez más importante y la cantidad de información que almacenan es cada vez mayor y más delicada, de ahí que la representación, manejo y transformación de la información es y continuará siendo en los próximos años, una carrera contra el tiempo.

Actividad de repaso

Instrucciones: Elabore programas en C++ para los siguientes problemas y también para los problemas del capítulo 1. Para los problemas nuevos, diseñe el algoritmo antes de hacer el programa.

Problema 1. Múltiplos

Para un algoritmo avanzado de cifrado de datos se requiere tener diferentes programas que resuelvan distintas relaciones entre números. Dado un par de números enteros, uno de estos programas debe determinar si uno de ellos es múltiplo del otro.

Ejemplos:

- Para los números 10 y 100 el programa debe indicar que 100 es múltiplo de 10.
- Para los números 3 y 4 el programa debe indicar que no existe relación.

Problema 2. Envasado de pintura

Una empresa morelense se encarga de producir y empaquetar pintura. Actualmente tiene varios contenedores con capacidad de hasta 100 litros y empaques comerciales estampados para venta de 20 litros, 6 litros y 1 litro de capacidad. Elabora un programa que, dada la cantidad de litros que tiene un contenedor, determine cuántos empaques de 20 litros pueden ser llenados, cuántos de 6 litros y cuántos de 1 litro. La prioridad es llenar primero todos los envases de 20 litros que posible; luego, los de 6 L.; y finalmente los de 1 L.

Problema 3. Calendarización

En una escuela se elaboran diferentes programas con la finalidad de automatizar la calendarización de los exámenes finales. De todos esos programas a ti te corresponde elaborar uno que le pida al usuario el día, mes y año correspondientes a

una fecha. Enseguida, el programa deberá indicarle al usuario si la fecha indicada es un día hábil o no, esto servirá para programar exámenes sólo de lunes a viernes.

Para elaborar esta solución necesitarás un conjunto de fórmulas que permiten determinar el día de la semana que corresponde a una fecha específica. Dados los valores de las variables day, month y year las fórmulas se muestran en la figura. El valor de d revela el día de la semana que corresponde a la fecha day/month/year. Para interpretar el resultado considera lo siguiente: si d es 0 significa domingo, si es 1 significa lunes y así sucesivamente.

$$a = \frac{14 - month}{12} \qquad \text{(mod es \%)}$$
$$y = year - a$$
$$m = month + 12a - 2$$
$$d = \left(day + y + \frac{y}{4} - \frac{y}{100} + \frac{y}{400} + \frac{31m}{12}\right) \bmod 7$$

NOTA: El valor de d va de 0 a 6 y los valores de day, month y year empiezan desde 1. Todas las variables involucradas en la fórmula deben ser de tipo int para que funcione.

Problema 4. Máquina de estacionamiento

En pocos meses se instalará un sistema electrónico para el pago de estacionamiento de la nueva plaza comercial de la ciudad. Se colocarán máquinas donde el usuario introduce su boleto y el dispositivo que lo recibe envía el dato a un programa que se encarga de calcular el cambio. Se requiere elaborar un programa que, por lo pronto, pida ambos datos del teclado: el importe del estacionamiento y monto que paga el cliente (luego, se unirá el programa con el dispositivo para que estos datos lleguen automáticamente). El programa sólo puede dar monedas de 10, 5, 2 y 1 peso y se requiere que el dispositivo entregue el menor número de monedas posible para alargar el tiempo requerido entre una carga de dinero y la siguiente.

Deberá indicar el cambio que va a dar, cuántas monedas de cada denominación y el total de monedas que entregará.

Problema 5. El día de la Patria

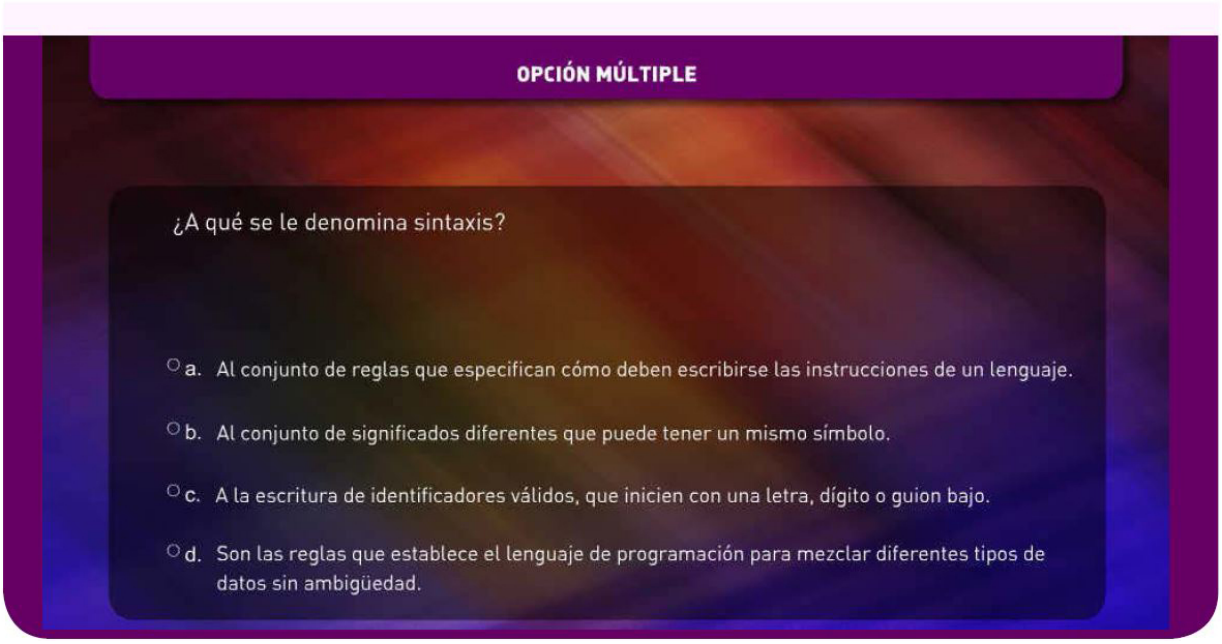
El gobierno de Cuernavaca acaba de instituir oficialmente el Día de la Patria. Los legisladores acordaron que sería el tercer sábado del mes de septiembre de cada año.

Elabora un programa que pida el año al usuario, enseguida deberá mostrar como respuesta la fecha en la que caería el Día de la Patria de ese año. Por ejemplo, si le damos 2009 como entrada, el programa deberá responder lo siguiente: “El Día de la Patria será el 19 de septiembre de 2009” (ver fórmula en el problema Calendarización).

Problema 6. Pares e impares

Elabora un programa que permita saber si un número dado es par o impar.

Ejercicio integrador del capítulo 2



Conclusión del capítulo 2

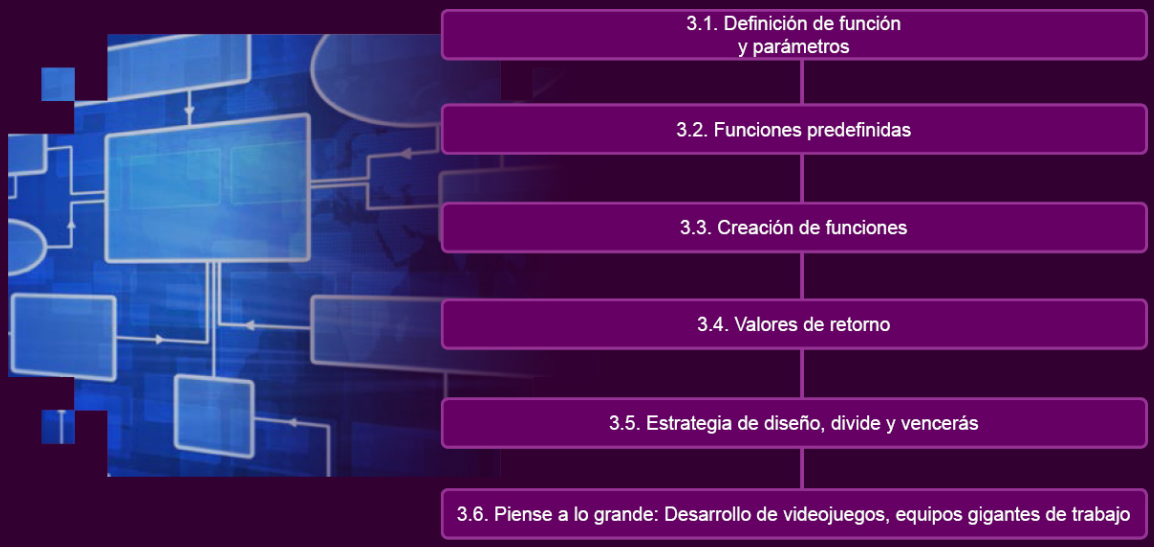
A lo largo de este capítulo se han podido dar los primeros pasos en el aprendizaje del lenguaje de programación C++. Ahora ha quedado claro que éste tiene reglas bien definidas y que cada uno de sus elementos conlleva un significado implícito. Al aprender programación, las personas con frecuencia ven en el lenguaje de programación una primer y difícil barrera. Tal y como ocurre cuando las personas enfrentan a un nuevo idioma.

Es útil asociar la programación con algo que resulte familiar. El símil más cercano son las matemáticas. Pero incluso cuando la notación es semejante, la programación es muy diferente de las matemáticas, a pesar de está basada en ellas. La computadora lleva a cabo un procesamiento numérico, secuencial y lógico, es decir, opera con datos concretos. En las matemáticas por su parte se razona de manera analítica utilizando símbolos matemáticos abstractos.

En conclusión, las matemáticas y la programación están íntimamente relacionadas y se complementan a la perfección, pero operan a diferentes niveles de razonamiento. Se ha visto que no hay números reales, sino tipos *float* y *double.* Mientras que los números reales son infinitos, los tipos *float* y *double* pueden representar sólo una cantidad finita de números. En la medida en que se entienda esta diferencia y se visualice la lógica con que opera una computadora, se empezará a ver el enorme potencial que se esconde tras esa aparente barrera de reglas y sintaxis.

Capítulo 3. Funciones: Organización y eficiencia

Organizador temático



Funciones: Organización y eficiencia

3.1 Definición de función y parámetros

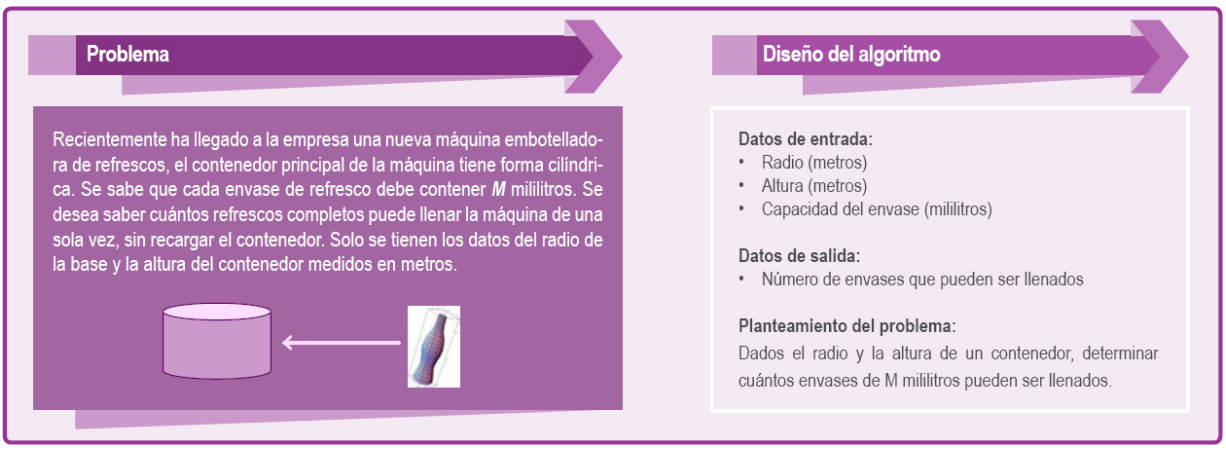
En la actualidad, la creación de software no es sólo un reto computacional, sino también de organización, distribución y gestión del trabajo pues, en los últimos años, se han creado un sinfín de herramientas y elementos de software, cada vez más complejos y sofisticados dada la problemática que resuelven. En el proceso de construcción de software de gran tamaño, participan decenas, cientos e incluso miles de personas.

Para la gestión de esta complejidad, los lenguajes de programación tienen un conjunto de mecanismos que permiten organizar de mejor manera el código que se escribe. Hasta ahora, los problemas que se han abordado son pequeños comparados con las grandes herramientas de software, pero conforme un programa crece, es necesario subdividirlo. Esto último permite mantener el control y la organización del proyecto, y delimitar con precisión cada uno de los sub problemas que deben afrontarse para resolver el problema general.

Al mismo tiempo, esta subdivisión permite distribuir el trabajo entre diferentes personas, cada una de ellas con una tarea específica, que una vez realizada, se une al trabajo de todos. Esto permite también la reutilización del código generado en la solución de problemas similares. Como muchos otros lenguajes, C++ dispone de un mecanismo para realizar esta descomposición de los problemas: las funciones.

Una función es un conjunto de instrucciones de programación, diseñadas para realizar una tarea específica dentro de un programa. Una función puede recibir información de entrada; procesarla y devolver información de salida. Esto permite a las funciones, operar de la misma forma en que lo hace el software: recibir información, procesarla y arrojar un resultado.

A continuación se revisa un ejemplo práctico para comenzar a identificar los criterios que deben emplearse para definir funciones en un programa. Como primer paso, revise la Figura 3.1 para identificar las diferentes etapas de la solución de un problema usando programación.



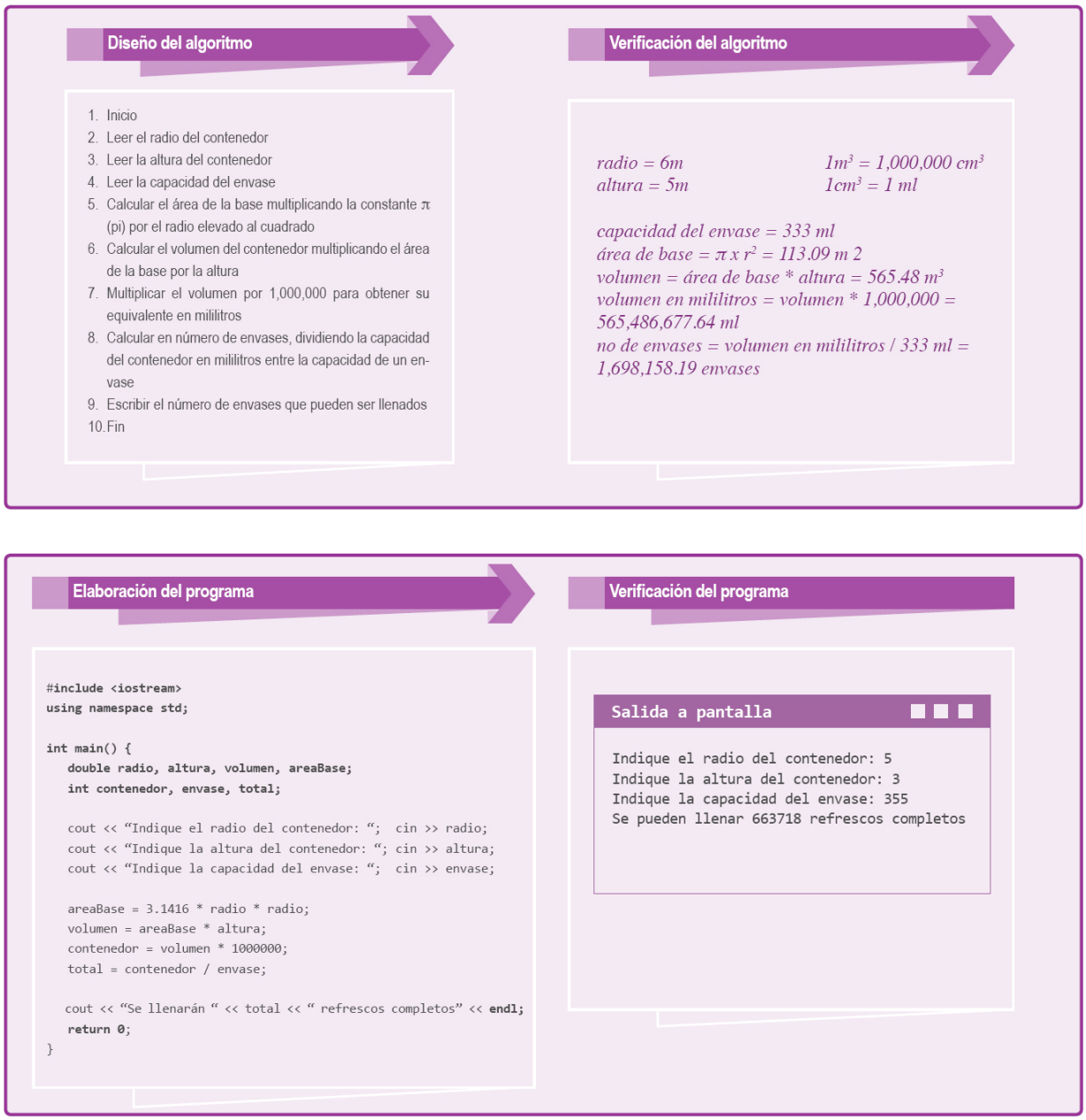


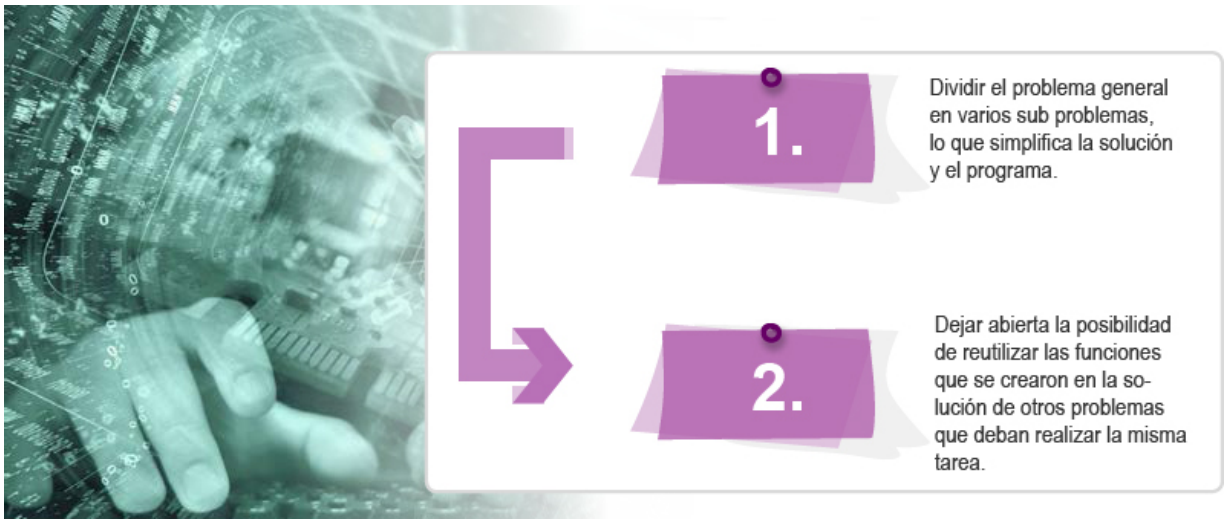
Figura 3. 1. Diferentes etapas para resolver problema de la embotelladora.

Si se analizan detenidamente el algoritmo y el código fuente, puede verse que aun en el caso de problemas pequeños como éste, hay sub problemas que deben resolverse para llegar a la solución general. Por ejemplo: calcular el volumen de un cilindro no es el objetivo del problema general, sin embargo es una tarea necesaria para determinar el número de refrescos completos que pueden llenarse. Realizar la conversión de metros cúbicos a mililitros

tampoco es el punto medular del problema, pero es igualmente necesario para determinar el número de refrescos completos que pueden llenarse.

Definir funciones en un programa, consiste en identificar tareas completas y específicas que puedan ser aisladas del programa principal sin que éste pierda sentido. Dichas tareas deben estar bien delimitadas, además de que debe identificarse claramente para cada una de ellas, las entradas y salidas correspondientes. Este análisis y diseño basado en funciones persigue dos objetivos muy importantes:

En la Figura 3.2 puede revisarse la solución del problema de la embotelladora con funciones. Observa que las funciones actúan a su vez como pequeños programas independientes, de ahí que este enfoque facilite el trabajo en equipo y mejore su organización.



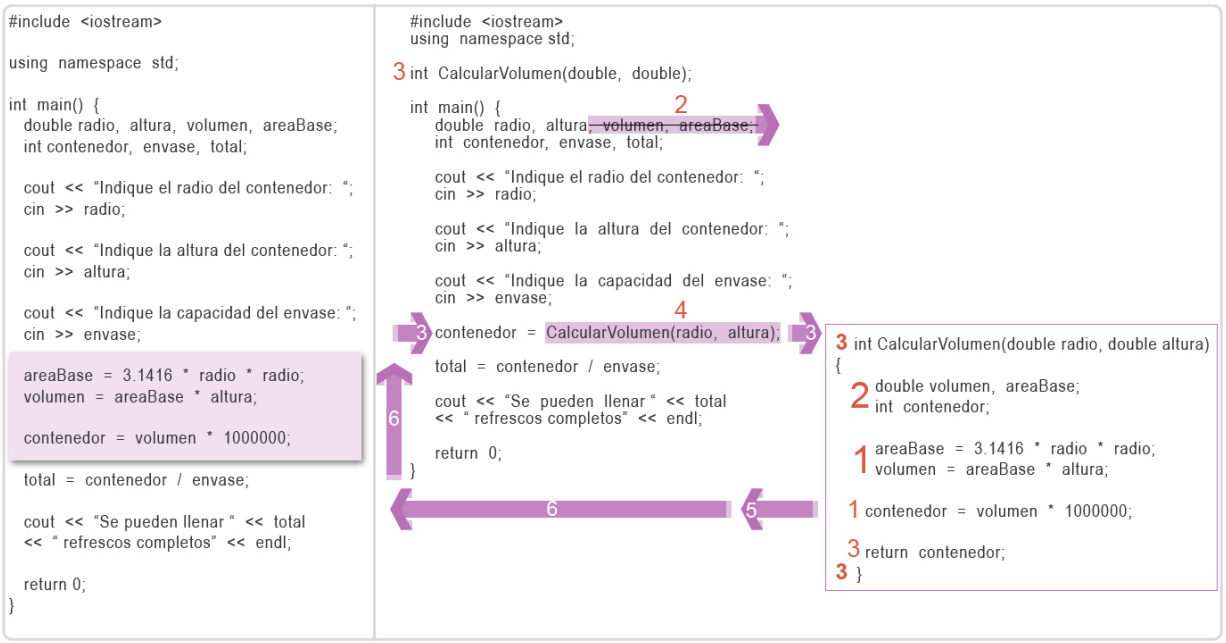


Figura 3. 2. Creación de una función a partir de un segmento de código.

En este punto es fundamental que el lector tenga una idea general del propósito y utilidad asociados con la definición de funciones. Observe, por ejemplo, que en la Figura 3.2, la función *Calcular Volumen* tiene asociado un tipo de dato. Éste corresponde con el tipo de dato que la función devuelve por medio de la instrucción *return.* De la misma forma, los **parámetros** *radio* y *altura* en la función son del mismo tipo y están en el mismo orden que las variables *radio* y *altura* en el programa principal. Es muy importante cuidar esta consistencia cuando se trabaja con funciones.

De manera similar al ejemplo de la embotelladora, pueden crearse funciones con diferente número y tipo de parámetros de entrada, dependiendo del sub problema particular que resuelvan. A continuación se revisarán algunos ejemplos de funciones predefinidas del lenguaje C++.

3.2 Funciones predefinidas

El usuario puede definir funciones propias, pero cada lenguaje de programación tiene predefinido un conjunto de funciones. En el caso de C++ se les denomina: biblioteca estándar. En el ámbito de la programación, se usa el término **biblioteca** para

denotar a un conjunto de funciones agrupadas según su uso, y que proporcionan la funcionalidad básica que la mayoría de los programas podrían necesitar.

Las bibliotecas estándar están cuidadosamente diseñadas y optimizadas, de tal manera que se recomienda utilizarlas siempre que se requiera esa funcionalidad. Se considera una mala práctica diseñar una función propia, cuando la biblioteca estándar ya proporciona una función que realiza la misma tarea.

En el nuevo estándar internacional C++11 (ISO/IEC 14882, 2011) se definen trece categorías para las bibliotecas estándar de C++. En la Tabla 3.1 se muestran sus nombres y una breve descripción de cada una.

Tabla 3.1. Categorías de las bibliotecas estándar de C++

Número	Nombre	Descripción
1	Biblioteca de soporte del lenguaje.	Manejo de memoria y excepciones.
2	Biblioteca de diagnóstico.	Reporte y detección de errores.
3	Biblioteca de utilidades generales.	Almacenamiento dinámico, estructuras y manejo de tiempo.
4	Biblioteca de Srings.	Manipulación de texto.
5	Biblioteca de localización.	Soporte de procesamiento de texto para internacionalización.
6	Biblioteca de contenedores.	Estructuras de datos más utilizadas.
7	Biblioteca de iteradores.	Métodos optimizados de acceso a estructuras de datos.
8	Biblioteca de algoritmos.	Algoritmos comunes.
9	Biblioteca numérica.	Manejo de números complejos y números aleatorios.
10	Biblioteca de entrada/salida.	Manejo de flujos(streams) de entrada y salida.

Número	Nombre	Descripción
11	Biblioteca de expresiones regulares.	Soporte de búsqueda y patrones basados en expresiones regulares.
12	Biblioteca de operaciones atómicas.	Manejo de acceso concurrente a datos compartidos.
13	Biblioteca de sopore de hilos.	Soporte de manejo de hilos, exclusión e intercamunicación.

Cada una de las categorías agrupa un conjunto de bibliotecas. Para tener acceso a las funciones de una biblioteca debe incluirse en el programa el encabezado correspondiente. Dichos encabezados se incluyen en los programas por medio de la *directiva include,* tal como se ha hecho con el encabezado *iostream* en el ejemplo de la embotelladora. En la Tabla 3.2 se muestran algunas de las bibliotecas más utilizadas. La lista completa de encabezados por categoría escapa al alcance de este eBook, pero puede consultarse en el estándar del lenguaje C++ publicado en septiembre de 2011 (ISO/IEC 14882, 2011). De manera adicional al documento de ISO, el lector también puede acceder al sitio *cppreference,* que está constantemente actualizado de acuerdo con el nuevo estándar.

Tabla 3.2. Algunas de las bibliotecas más utilizadas de la biblioteca estándar de C++

Biblioteca	Descripción
cstdlib	Funciones básicas para el manejo de la aplicación.
ctime	Funciones para manipulación de fecha y hora.
string	Funciones para el manejo de cadenas de caracteres.
array	Funciones para el manejo de arreglos.
vector	Funciones para el manejo de vectores.
algorithm	Funciones de ordenamiento y búsqueda.

Biblioteca	Descripción
cmath	Funciones matemáticas comunes.
iostream	Funciones de flujo de entrada y salida.

A continuación se presenta un ejemplo sobre el uso de funciones predefinidas en C++. Use la Figura 3.3 para observar la utilidad de las funciones matemáticas en la solución de un problema. Observe que se incluye el encabezado *cmath* para utilizar dichas funciones.

Problema

Dados un robot móvil que se encuentra navegando en un espacio abierto y las posiciones de dos puntos de carga de batería: elabore un programa que determine a qué distancia del robot se encuentra el punto de carga más cercano. Asuma que se conoce también la posición del robot en el espacio.

Análisis del problema

Datos de entrada:

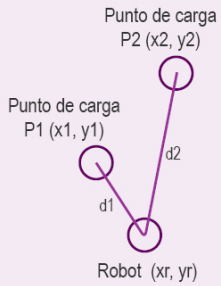
- Posición del robot (xr, yr)
- Posición del punto de carga 1 (x1, y1)
- Posición del punto de carga 2 (x2, y2)

Datos de salida:

- Distancia a la que se encuentra el punto de carga más cercano

Planteamiento del problema:

Dadas las posiciones de dos puntos en el espacio y la posición de un robot, determinar cuál de los dos puntos está más cerca del robot.



Diseño del algoritmo

1. Inicio
2. Leer las coordenadas del robot R (xr,yr)
3. Leer las coordenadas del punto de carga P1 (x1,y1)
4. Leer las coordenadas del punto de carga P2 (x2,y2)
5. Calcular la distancia **d1** entre el robot **R** y el punto de carga **P1**. Para calcularla se obtiene la raíz cuadrada de la suma de los cuadrados de las diferencias de los componentes (x, y) de **R** y **P1**.
6. Calcular la distancia **d2** entre el robot **R** y el punto de carga **P2**. Para calcularla se obtiene la raíz cuadrada de la suma de los cuadrados de las diferencias de los componentes (x, y) de **R** y **P2**.
7. Calcular el valor mínimo de **d1** y **d2**.
8. Escribir el valor mínimo
9. Fin

Verificación del algoritmo

R (xr, yr) = (1,0)
P1 (x1, y1) = (-2,2)
P2 (x2, y2) = (3,7)

d1 = 3.60555
d2 = 7.28011
mínimo = 3.60555

La distancia al punto más cercano es 3.60555

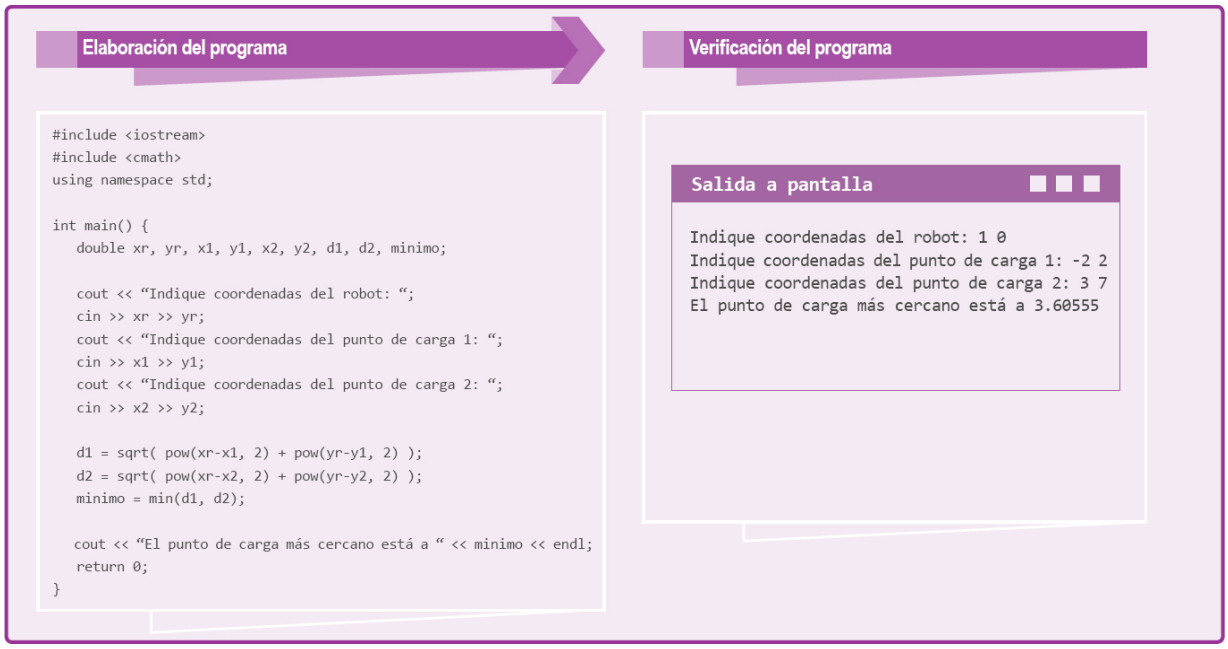


Figura 3. 3. Solución al problema del robot y los puntos de carga.

Como sucede con frecuencia, no existe una única manera de resolver un problema, también podría haberse usado la función hipotenusa, *hypot,* en lugar de usar la función raíz cuadrada, *sqrt,* y la función potencia, *pow.* Observe a continuación la definición de la función hypot.

```
// Calcula la hipotenusa del triángulo dados los catetos a y b.

double hypot(double a, double b);
```

La definición de una función es la especificación de sus características. Estas características son: el tipo de retorno, el nombre de la función y el número y tipo de los parámetros. Siempre que se quiera hacer uso de una función debe consultarse su definición.

De ahí que el siguiente código:

```
d1 = sqrt( pow(xr-x1, 2) + pow(yr-y1, 2) );
```

```
d2 = sqrt( pow(xr-x2, 2) + pow(yr-y2, 2) );
```

Puede ser remplazado por el código equivalente:

```
d1 = hypot(xr-x1, yr-y1);

d2 = hypot(xr-x2, yr-y2);
```

Cabe recalcar que el número y tipo de parámetros debe respetarse al hacer uso de una función, sin embargo, esto no impide colocar operaciones aritméticas como argumentos, ya que antes de hacer la llamada a la función, la computadora evaluará la expresión y enviará el valor resultante al parámetro correspondiente. Se denomina parámetros a aquellos símbolos empleados en la definición general de la función, mientras que los argumentos son datos o identificadores específicos con los que se ejecuta una función en un instante dado.

En el siguiente problema se utilizan dos funciones de la biblioteca cstdlib y una de la biblioteca ctime. Las dos primeras son srand() y rand(), funciones que se utilizan para inicializar el **generador de números pseudoaleatorios** y para generar un número aleatorio, respectivamente. La tercera es time(), que devuelve el número de segundos que han transcurrido desde el 1 de enero de 1970, hasta el momento de la llamada a la función. Esto último es particularmente útil, ya que se puede usar el valor que devuelve time() como **semilla** para srand(). Semillas diferentes arrojan secuencias de números aleatorios diferentes, por lo que cada vez que se ejecute el programa la función time() arrojará un valor distinto, cambiando con ello la secuencia de aleatorios generada en cada ejecución. En la Figura 3.4 se observa el uso de las funciones mencionadas en el contexto de la solución a un problema.

Problema

Elabore un programa que simule un juego de apuestas entre la computadora y el usuario. Para el juego se utiliza un par de dados y la mecánica es la siguiente: el programa debe simular la tirada de un par de datos por parte de la computadora y mostrar el resultado en pantalla. Enseguida el usuario debe evaluar sus posibilidades de obtener un resultado mayor al de la computadora, indicar el monto de la apuesta y presionar enter. El programa debe simular la tirada del usuario e indicar si pierde o gana la apuesta.

Análisis del problema

Datos de entrada:

- El monto de la apuesta

Datos de salida:

- La simulación de la tirada de dos dados de la computadora
- La simulación de la tirada de dos dados del usuario
- El mensaje que indique si el usuario gana o pierde la apuesta.

Planteamiento del problema:

Dadas dos simulaciones de la tirada de un par de dados: uno de la computadora y otro del usuario, determinar el ganador de una apuesta dependiendo de cuál de los dos valores es el mayor.

Diseño del algoritmo

Consideraciones de diseño:

- Necesitamos generar números aleatorios para simular la tirada de cada dado como un evento al azar.
- Un dado tiene seis posibles resultados, de 1 a 6.
- La definición de la función rand indica que genera un número entre 0 y RAND_MAX
- RAND_MAX tiene un valor de 2,147,483,647 (puede variar dependiendo de la implementación de C++ en cada plataforma)
- Para acotar el rango de 1 a 6 se puede usar el operador módulo y luego sumarle 1 al resultado.
- Una vez calculados ambos dados se suman y se almacena el resultado de la computadora y del usuario.
- El mayor es el que gana la apuesta.

Diseño del algoritmo

```
Inicio
  dado1 = (rand() % 6) + 1
  dado2 = (rand() % 6) + 1
  sumaCompu = dado1 + dado2
  Escribir sumaCompu
  Leer apuesta
  dado1 = (rand() % 6) + 1
  dado2 = (rand() % 6) + 1
  sumaHumano = dado1 + dado2
  Escribir sumaHumano
  si (sumaHumano > sumaCompu) entonces{
      Escribir "El humano gana la apuesta"
  }de lo contrario{
      Escribir "La computadora gana la apuesta"
  }
fin
```

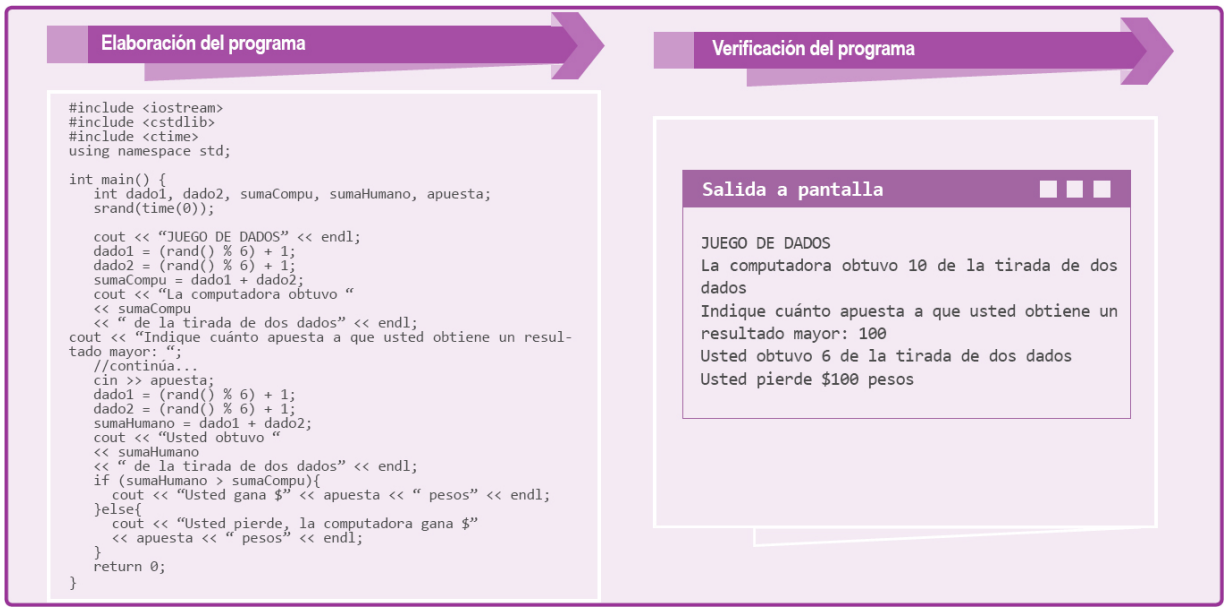


Figura 3. 4. Simulación de un juego usando dados

Como ya se mencionó el lenguaje C++ permite al programador definir sus propias funciones, ya sea para subdividir el problema o con el propósito de permitir su reutilización. En seguida se mostrarán las reglas y los criterios para la creación de funciones.

3.3 Creación de funciones

La creación de funciones es una necesidad recurrente en la programación, de hecho ni siquiera se podría concebir una solución compleja sin la separación y granularidad que las funciones proporcionan.

Al igual que las variables las funciones tienen un tipo de dato asociado, que se relaciona íntimamente con la instrucción return. Esta instrucción devuelve el control del programa y el valor de retorno al punto donde fue llamada la función.

Enseguida del nombre de la función deben indicarse los parámetros que recibe como entrada. Estos últimos deben escribirse separados por comas e indicar el tipo de dato y el nombre de cada uno.

El nombre que se ponga a los parámetros, será el usado para referirse a ellos dentro de la función. Es importante recalcar que su ámbito está confinado al bloque de la función, por lo que todo parámetro forma parte de las variables locales de la misma. En la Figura 3.5 se identifican las partes que componen la definición de una función:

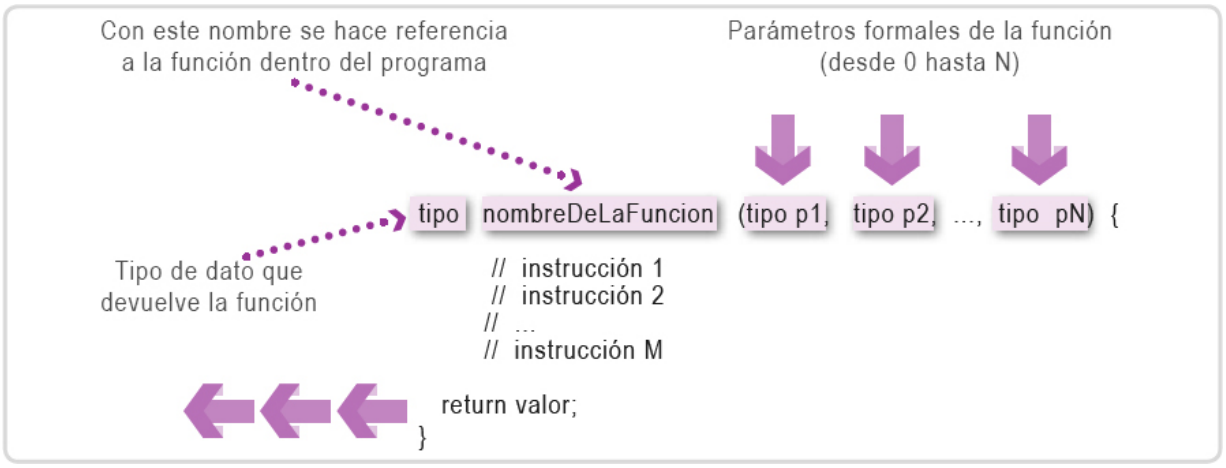


Figura 3. 5. Partes que componen la definición de una función.

Cabe destacar que, al igual que las variables, las funciones deben tener un nombre significativo, relacionado con el procedimiento que realizan. Esto permitirá una fácil comprensión del código ya sea propio o ajeno. Es común que los nombres de las funciones incluyan un verbo, pues se relacionan con acciones específicas.

De manera complementaria a la definición de la función está el llamado *prototipo* de la función que indica al compilador las características de una función, por lo que se coloca antes de invocar a la función misma. Su estructura es semejante a la definición de la función, con la diferencia de que no se colocan los nombres de los parámetros, ni el cuerpo de la función. Esto se debe a que el compilador sólo necesita validar que el nombre, tipo y número de parámetros sea el correcto. Para el caso de la función mostrada en la Figura 3.4 el prototipo quedaría como sigue:

```
tipo nombreDeLaFuncion(tipo, tipo, ..., tipo);
```

En la Figura 3.6 el lector puede verificar si ha comprendido correctamente los conceptos asociados a las funciones:

Código en vivo

```
#include <iostream>
using namespace std;

double promediar(int, int, int);

int main( ) {
      double resultado;
      resultado = promediar(90, 100, 91);
      cout << "El promedio es: " << resultado << endl;
      return 0;
}


double promediar(int a, int b, int c
      ){
      double promedio;
      promedio = (a + b + c) / 3.0;
      return promedio;
}
```

Salida a pantalla

```
El promedio es: 93.6667
```

Figura 3. 6. Ejemplo del uso y definición de funciones y parámetros.

A continuación se aborda un ejemplo de la definición de una función como parte de la solución a un problema. En la Figura 3.7 puede analizarse el papel de las funciones en la solución a un problema específico:

Problema

Al interior de una organización el director general ha establecido tener reuniones masivas con todos los empleados de la planta el primer lunes de cada trimestre. Elabore un programa que dado el año, calcule automáticamente las fechas de las reuniones trimestrales de todo el año. Use las fórmulas conocidas para determinar el día de la semana de una fecha day/month/year dada. El valor d es el día de la semana: 0 significa domingo, 1 lunes y así sucesivamente.

$$a = \frac{14 - month}{12} \qquad \text{(mod es el operador \%)}$$

$$y = year - a$$

$$m = month + 12a - 2$$

$$d = (day + y + \frac{y}{4} - \frac{y}{100} + \frac{y}{400} + \frac{31m}{12}) \bmod 7$$

Diseño del algoritmo

Consideraciones de diseño:
Del planteamiento podemos ver que se debe realizar cuatro veces la misma tarea: determinar la fecha del primer lunes del mes.
Lo anterior debe hacerse para enero, abril, julio y octubre
Podemos definir una función que reciba el mes y el año para usarla cuatro veces.
Un punto de partida es el día de la semana que corresponde al día 1 del mes, para de ahí calcular la fecha del primer lunes. Revisemos los casos:

Dia de la semana	Fecha del primer lunes
0 (domingo)	1+1 (1 día después del 1)
1 (lunes)	1+0 (0 días después del 1)
2 (martes)	1+6 (seis días después del 1)
3 (miércoles)	1+5 (cinco días después del 1)
4 (jueves)	1+4 (cuatro días después del 1)
5 (viernes)	1+3 (tres días después del 1)
6 (sábado)	1+2 (dos días después del 1)

Diseño del algoritmo

- De los últimos 5 casos pareciera que se trata de un complemento a 8, es decir: fecha = 1 + (8 -día de la semana)
- Sin embargo los dos primeros casos darían como resultado 8 y 7 respectivamente. Para obtener los valores deseados se recurre al operador módulo quedando de la siguiente manera: fecha = 1 + (8 -día de la semana) % 7
- Revisamos los casos y verificamos que es totalmente consistente con los valores observados.

Diseño del algoritmo

```
Inicio
   Leer anio
   fecha1 = calcularPrimerLunes(1, anio);
   fecha2 = calcularPrimerLunes(4, anio);
   fecha3 = calcularPrimerLunes(7, anio);
   fecha4 = calcularPrimerLunes(10, anio);
   Escribir “Fecha de la primer reunión: “ fecha1 “ de enero”
   Escribir “Fecha de la segunda reunión: “ fecha2 “ de abril”
   Escribir “Fecha de la tercer reunión: “ fecha3 “ de julio”
   Escribir “Fecha de la cuarta reunión: “ fecha4 “ de octubre”
Fin
```

Elaboración del programa

```
#include <iostream>
using namespace std;
int calcularPrimerLunes(int, int);

int main() {
   int anio, fecha1, fecha2, fecha3, fecha4;

   cout << “Indique el año: “;
   cin >> anio;

   fecha1 = calcularPrimerLunes(1,anio);
   fecha2 = calcularPrimerLunes(4,anio);
   fecha3 = calcularPrimerLunes(7,anio);
   fecha4 = calcularPrimerLunes(10,anio);
   cout << “Fecha de primer reunión: “ << fecha1 << “ de enero” << endl;
   cout << “Fecha de segunda reunión: “ << fecha2 << “ de abril” << endl;
   cout << “Fecha de tercer reunión: “ << fecha3 << “ de julio” << endl;
    cout << “Fecha de cuarta reunión: “ << fecha4 << “ de octubre” <<
endl;

   return 0;
}
```

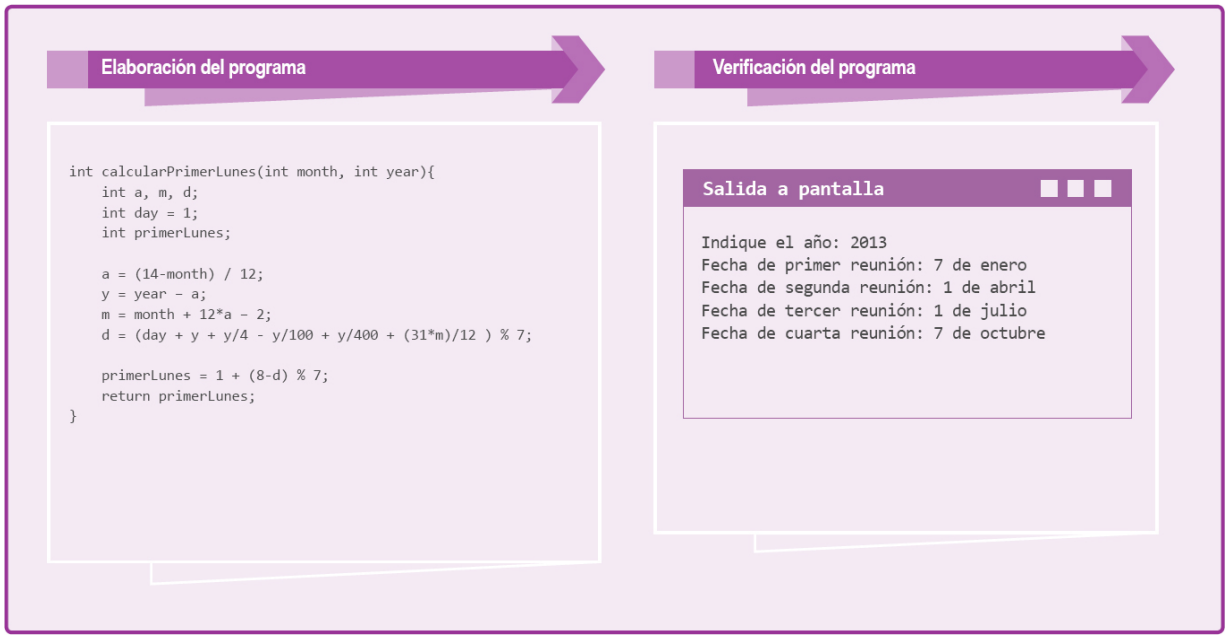


Figura 3. 7. Ejemplo de la definición de una función en la solución a un problema.

Ya sea con el objetivo de favorecer la reutilización del código o para subdividir un problema, las funciones proporcionan el mecanismo necesario para segmentar el código, ventaja particularmente útil en equipos de trabajo integrados por decenas, cientos o miles de personas, quienes con frecuencia están distribuidos geográficamente y requieren de un esquema bien estructurado para trabajar de manera colaborativa.

Paso de parámetros por referencia

Hasta este momento, todos los ejemplos de funciones que se han abordado reciben parámetros por copia. Es decir, se envía una copia del dato contenido en la variable usada como argumento de la función. Esto implica que cualquier cambio efectuado sobre la variable local dentro de la función no afecta a la variable utilizada como argumento.

De acuerdo con el problema específico podría ser necesario pasar un parámetro por referencia, es decir, enviar la **dirección de memoria** de la variable. De esta forma, cualquier cambio hecho a la variable dentro de la función se reflejará en la variable original.

El paso de parámetros por referencia se indica por medio del caracter & (ampersand). Este caracter se coloca después del tipo de dato en el prototipo de la función y precede al nombre del parámetro en la definición de la función.

Para comprender la diferencia entre el paso de parámetros por valor y el paso de parámetros por referencia revise la Figura 3.8.

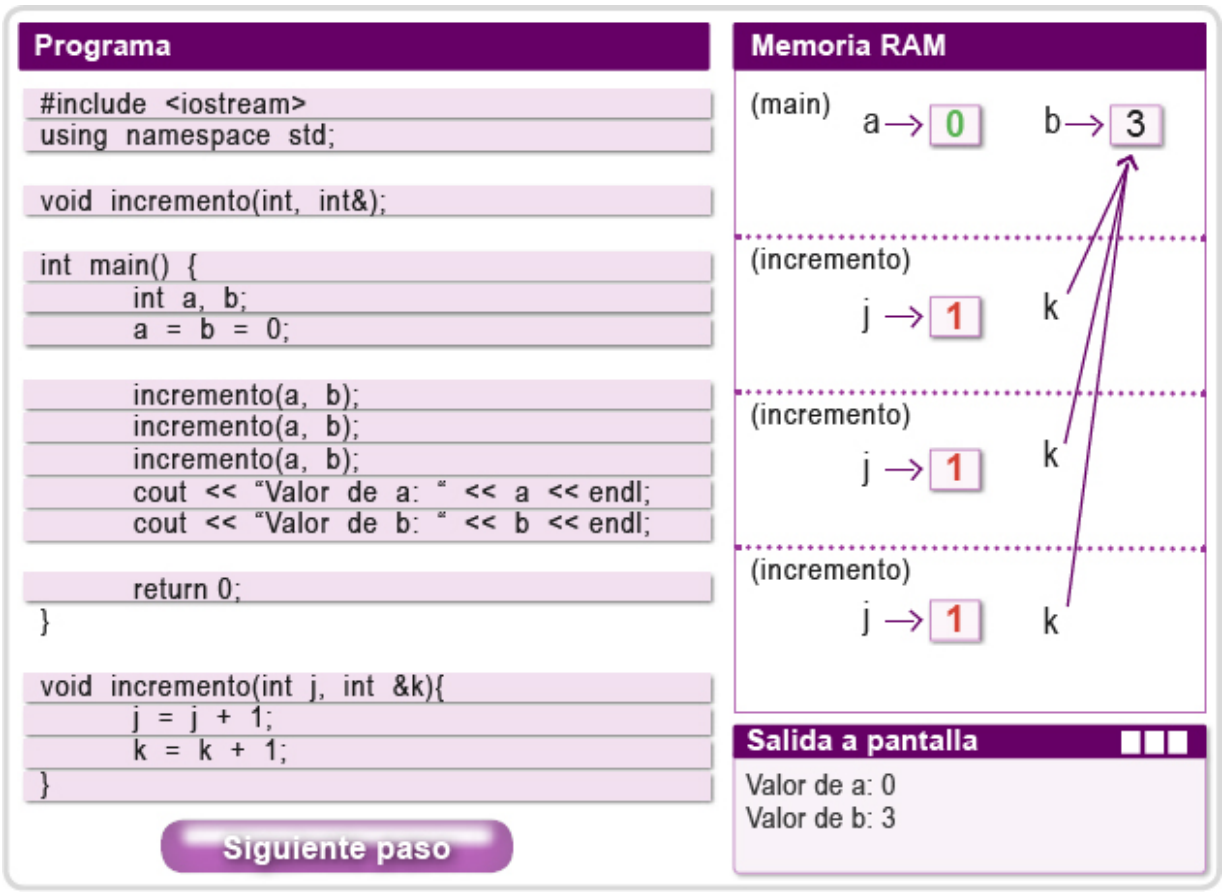


Figura 3. 8. Paso de parámetros por valor y por referencia.

En la Figura 3.9 puede revisarse la solución a un problema específico y el uso del paso de parámetros por valor y por referencia.

Problema

En una competencia olímpica de clavados. un total de cinco jueces califican a cada uno de los competidores. Se requiere diseñar un programa que reciba las calificaciones de los jueces, elimine la más alta, la más baja y presente el promedio de las tres restantes.

Análisis del problema

Datos de entrada:

- Las cinco calificaciones de los jueces.

Datos de salida:

- El promedio de las tres calificaciones restantes, quitando la más baja y la más alta.

Planteamiento del problema:

Dados cinco números, determinar el promedio de los tres números restantes después de eliminar el menor y el mayor.

Diseño del algoritmo

Consideraciones de diseño:

- Se requiere identificar una manera de obtener el menor
- Se requiere identificar una manera de obtener el mayor
- En ambos casos:
  - Puede hacerse usando cinco variables diferentes, una por cada calificación.
  - Puede obtenerse el mayor comparando pares de ellas, pero eso llevaría a realizar muchas comparaciones y condiciones.
  - Puede hacerse incrementalmente. Es decir, ir obteniendo el mayor y el menor conforme se recibe una nueva calificación.
- Se requiere usar dos variables, una para almacenar el menor valor y otra para el mayor.
- El valor inicial de las variables puede ser la primer calificación introducida, e ir cambiando el valor conforme se van procesando el resto de las calificaciones.

Diseño del algoritmo

```
Inicio
   Leer c1  // primer calificación
   menor = c1
   mayor = c1
   Leer c2  // segunda calificación
   actualizarMayorMenor(c2, mayor, menor)
   Leer c3  // tercer calificación
   actualizarMayorMenor(c3, mayor, menor)
   Leer c4  // cuarta calificación
   actualizarMayorMenor(c4, mayor, menor)
   Leer c5  // quinta calificación
   actualizarMayorMenor(c5, mayor, menor)
   promedio = (c1 + c2 + c3 + c4 + c5 – menor – mayor) / 3
   Escribir promedio
Fin
```

Diseño del algoritmo

```
void actualizarMayorMenor(calif, &maximo, &minimo)
Inicio
   // maximo es una variable pasada por referencia
   maximo = max(maximo, calif)
   // minimo es una variable pasada por referencia
   minimo = min(minimo, calif)
Fin
```

Elaboración del programa

```
#include <iostream>
#include <cmath>
#include <string>
using namespace std;

void actualizarMaximoMinimo(double, double&, double&);
double leerDato(string);

int main() {
     double mayor, menor, c1, c2, c3, c4, c5, promedio;
     c1 = leerDato("Calificación del juez 1: ");
     mayor = menor = c1;
     c2 = leerDato ("Calificación del juez 2: ");
     actualizarMaximoMinimo(c2, mayor, menor);
     c3 = leerDato ("Calificación del juez 3: ");
     actualizarMaximoMinimo(c3, mayor, menor);
     c4 = leerDato ("Calificación del juez 4: ");
     actualizarMaximoMinimo(c4, mayor, menor);
     c5 = leerDato ("Calificación del juez 5: ");
     actualizarMaximoMinimo(c5, mayor, menor);
     promedio = (c1 + c2 + c3 + c4 + c5 - mayor - menor) / 3;
     cout << "El promedio es: " << promedio << endl;
     return 0;
}
```

Elaboración del programa

```
Elaboración del programa

double leerDato(string mensaje){
     double valor;
     cout << mensaje;
     cin >> valor;

     return valor;
}

void actualizarMaximoMinimo(double calif, double &maximo, dou-
ble &minimo){
     maximo = fmax(maximo, calif);
     minimo = fmin(minimo, calif);
}
```

Verificación del programa

Salida a pantalla

```
Calificación del juez 1: 8.5
Calificación del juez 2: 9.6
Calificación del juez 3: 10.0
Calificación del juez 4: 9.0
Calificación del juez 5: 9.0
El promedio es: 9.2
```

Figura 3. 9. Solución al problema de los jueces olímpicos

Es importante que el lector observe el uso de las dos funciones utilizadas en la solución de este problema. Por un lado, la función leerDato se utiliza para evitar escribir varias veces el mismo código. Esta función recibe un parámetro de tipo *string* que representa una cadena de caracteres alfanuméricos (ver capítulo 7) y devuelve un

valor de tipo *double.* De esta forma se evita tener que escribir cinco veces cout y cin.

Por otro lado, la función llamada actualizar-MaximiMinimo recibe tres parámetros: la calificación, una referencia a la variable mayor y una referencia a la variable menor. Dado que las dos últimas son direcciones de memoria, si su valor se modifica dentro de la función, el cambio persiste aun fuera de ella.

El uso de parámetros por valor o por referencia depende del diseño del algoritmo y es una poderosa herramienta para hacer más eficiente el uso de las funciones. Del dominio y la comprensión de este tema depende, en gran medida, el uso adecuado de las funciones.

3.4 Valores de retorno

Como se mencionó anteriormente, el valor de retorno es aquel que la función devuelve por medio de la instrucción return. Sin embargo, es posible también definir funciones que no devuelven ningún valor, es decir, que están diseñadas para realizar una tarea que no implica un resultado específico.

Un ejemplo de lo anterior podría ser el caso de una función que presenta datos en pantalla. La función puede definirse y utilizarse, pero, una vez enviada la información a la salida estándar la tarea ha terminado y no hay ningún valor que devolver. Para estos casos existe un tipo de dato especial denominado *void.*

La palabra void significa sin tipo. En el caso de una función void significa sin valor de retorno. Observe la definición de una función sin valor de retorno que permite dar formato a una cantidad y enviarla a la salida estándar:

```
  void mostrarPrecio(double p){

        cout << “$” << fixed << setprecision(2) << setw(10)
<< p << endl;
```

```
}
```

En este caso es claro que no hay ningún valor que devolver, la función se creó con la única finalidad de utilizarla varias veces para dar formato a montos económicos; además de lo anterior, la función aísla la solución del formato de la salida. Si el formato de la salida debe cambiar, basta con modificar la función mostrarPrecio y todas las salidas automáticamente respetarán la nueva convención.

3.5 Estrategia de diseño, divide y vencerás

Al diseñar algoritmos, es común identificar diferentes maneras de subdividir el problema. Esta división no solamente establece una partición lógica del trabajo, también busca reducir la complejidad ya que cada uno de los sub problemas debe ser más simple que el problema original. En la barra lateral podrás consultar el video y Anexo 3, donde se analizan ejemplos de esta estrategia de diseño de algoritmos, cabe mencionar que se usan condicionales por lo que si el lector no está familiarizado este concepto se le recomienda revisar previamente el capítulo 4.

3.6 Piense en grande: desarrollo de videojuegos, equipos gigantes de trabajo

De la gran diversidad de proyectos en los que se puede trabajar, el desarrollo de videojuegos es de los más difíciles de gestionar ya que a este tipo de proyectos se asocian varios factores críticos al mismo tiempo:

1. Un alto número de personas involucradas.
2. Una gran variedad de roles y perfiles profesionales involucrados.
3. Alta complejidad en los algoritmos que se deben programar.
4. Cambios constantes durante el proceso de desarrollo.

Lo anterior exige de estos proyectos el más alto grado de modularidad, la mayor claridad posible y una efectiva división del problema en varios niveles. Revise en la barra lateral un video sobre lo que los integrantes de un equipo de desarrollo de videojuegos viven como parte del día a día.

Como puede apreciarse cuando se estructura y se organiza un proyecto se establecen, en buena medida, las bases para el éxito o fracaso del mismo. Las funciones no son el único mecanismo del que disponen los lenguajes de programación para este fin, sin embargo, todos los mecanismos están enfocados al mismo objetivo: gestionar la complejidad del desarrollo de una solución, dividiendo el trabajo y simplificando a la vez el mantenimiento y detección de errores.

Actividad de repaso

Instrucciones: Analice los siguientes problemas y elabore las soluciones correspondientes.

Problema 1. Numeros pares

Elabore una función que permita determinar si un número dado es par.

Problema 2. Cálculos contables

Elabore tres funciones que dado el régimen en que se encuentra un contribuyente, calcule el ISR, el IVA y el bono por puntualidad, éste último de manera directamente proporcional al porcentaje de días que llegó a tiempo a trabajar.

Problema 3. Redondeo

Elabore una función que dado un número real, calcule número entero que corresponde al valor redondeado del dato de entrada.

Problema 4. Valor mínimo

Elabore una función que determine el menor de tres números.

Problema 5. Números aleatorios

Elabore una función que genere valores aleatorios dentro de un rango específico. La función debe recibir el valor mínimo del rango y el valor máximo del rango.

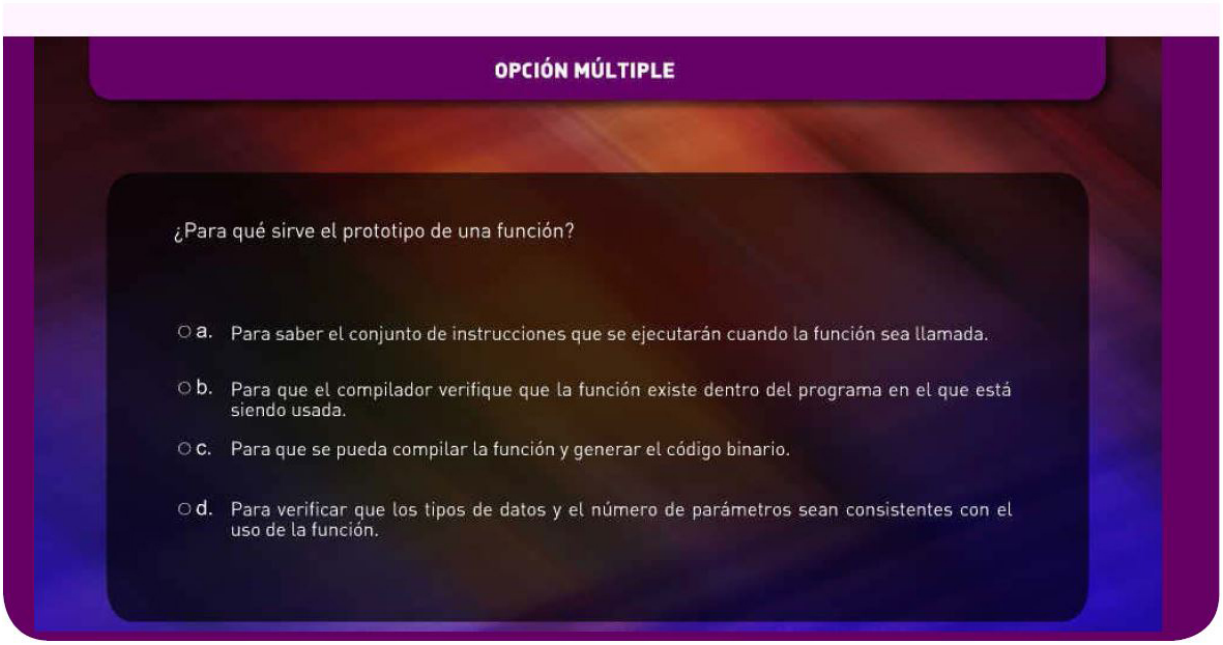
Problema 6. MCD

Elabore una función recursiva que calcule el máximo común divisor utilizando el método de Euclides.

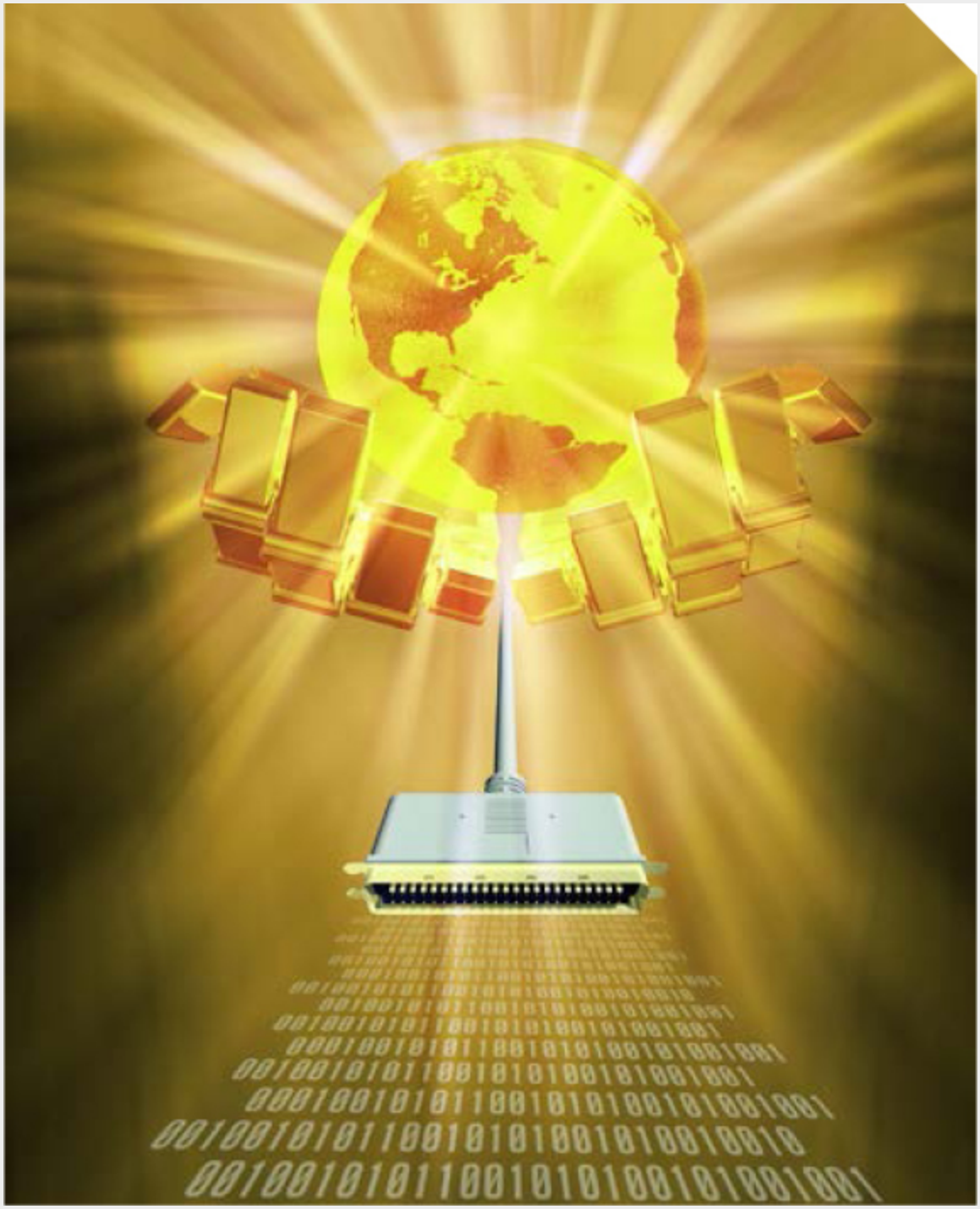
Problema 7. Permutaciones

Elabore una función recursiva que muestre todas las posibles permutaciones diferentes de números de 4 dígitos.

Ejercicio integrador del capítulo 3



Conclusión del capítulo 3



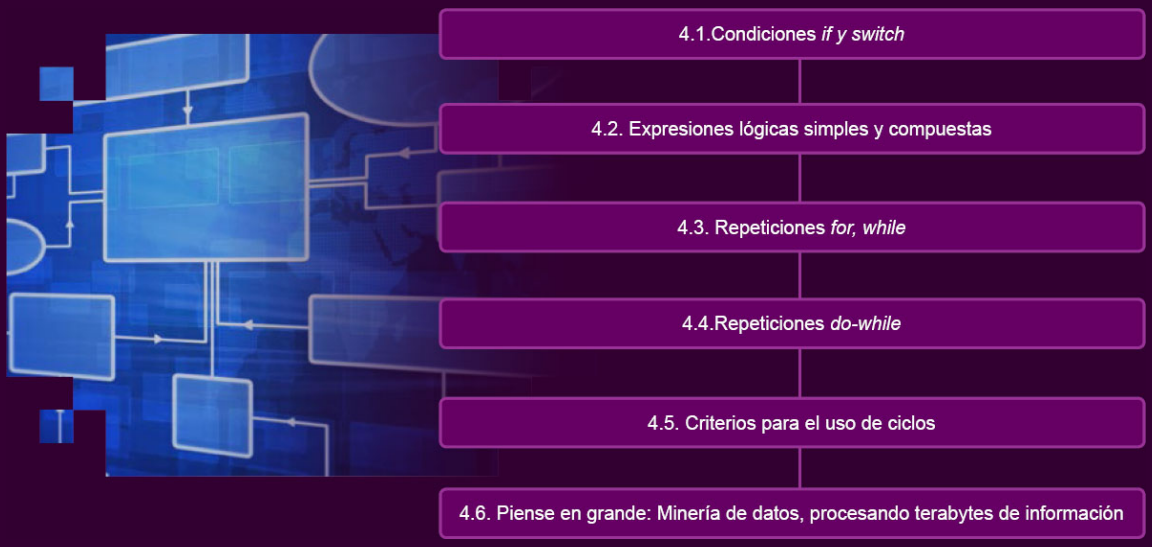
racias a los avances en el área de las ciencias computacionales, las herramientas que se desarrollan en la actualidad son cada vez más complejas y los tiempos de desarrollo se han vuelto más cortos. Esto lleva a las empresas a conformar equipos integrados por un mayor número de personas, lo que ha traído como consecuencia el incremento la complejidad de los proyectos y ha reforzado la necesidad de una mejor organización y división del trabajo.

En este capítulo se abordó uno de los mecanismos más simples para la división y organización del trabajo: las funciones. Éstas a su vez pueden organizarse en bibliotecas, que agrupan a las funciones según su propósito.

Del adecuado uso de estos mecanismos, depende la facilidad con que pueden unirse las contribuciones al software provenientes de distintas personas. De ahí que la planeación y organización, como en muchas otras áreas, resulte un factor de altísima prioridad en el proceso de solución de problemas con programación.

Capítulo 4. Estructuras de control

Organizador temático



Estructuras de control

4.1 Condiciones *(if y switch)*

En el proceso de desarrollo de soluciones computacionales algunas veces es necesario programar a la computadora para que ésta sea capaz de tomar decisiones. De manera similar a como lo hace una persona, un programa es capaz de analizar datos e información y decidir el mejor camino a seguir entre dos o más alternativas. Esta es una capacidad de enorme trascendencia en el ámbito de las ciencias computacionales. En consecuencia, los programas son capaces de analizar cada vez más alternativas de acción a fin de presentar soluciones eficientes y robustas a los problemas actuales de las personas y de las empresas.

A las instrucciones empleadas para la toma de decisiones con base en información disponible se les denomina condiciones o condicionales.

Un aspecto clave es que cada vez que se utiliza una instrucción condiciona, aumenta el número de posibles rutas de ejecución en un programa. Es decir, cada condicional incrementa el número de comportamientos diferentes que ofrece un programa. En la Figura 4.1 se ilustra el concepto de las condicionales con la intención de que el lector comprenda el funcionamiento de las mismas dentro de un programa:

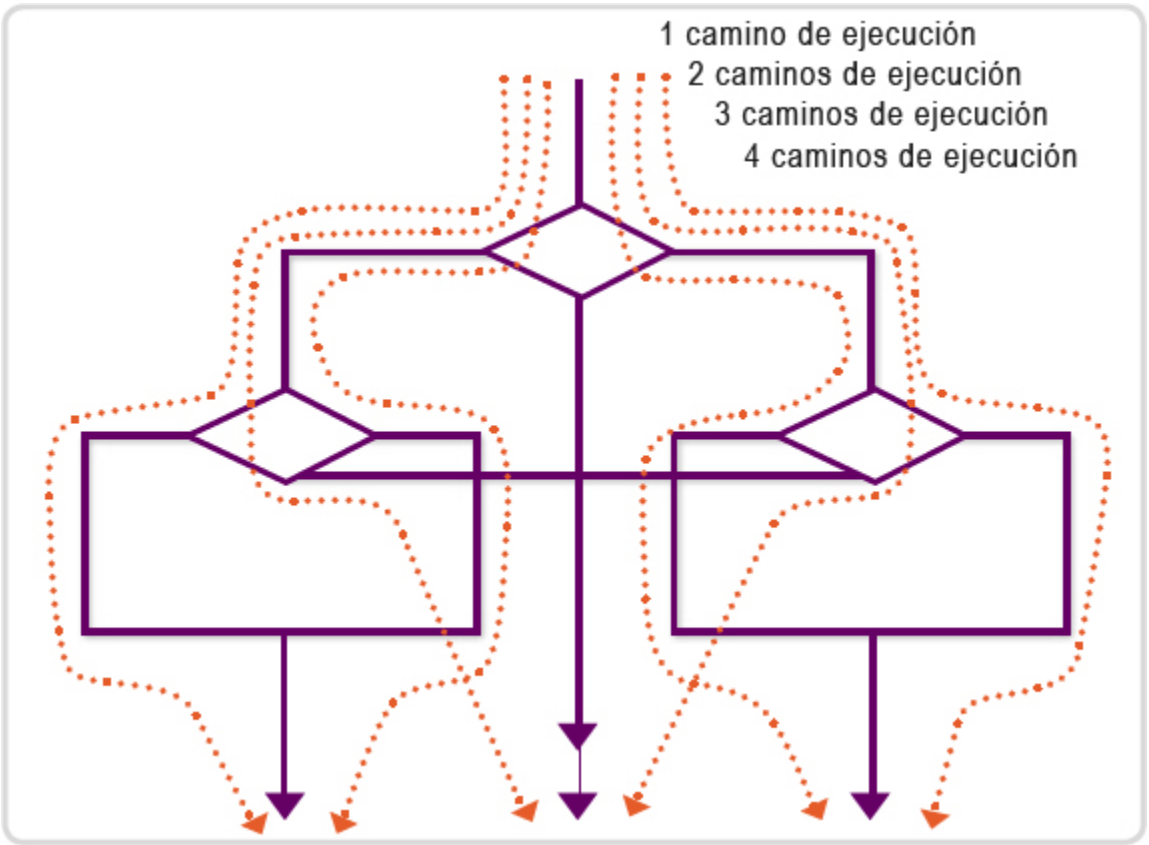


Figura 4.1. Concepto de condicionales dentro de un programa.

Ante una bifurcación en la ejecución de un programa, la computadora debe determinar el camino a seguir con base en la información existente. Para lo cual es necesario definir una expresión lógica, es decir, una expresión que genere como resultado

un **valor de verdad**. De esta forma, la computadora podrá decidir el camino a seguir dependiendo del resultado de la expresión lógica.

Código en vivo

```
int alu, a, b, c;
float precio, x, y, z;
char c;

if ( a > b > c ) ...
if ( a > 300 || a < 100 ) ...
if ( precio*2 = 3 ) ...
if ( alu < 12 ) ...
if ( b >= 1000 ) ...
if ( c % 2 == 1 ) ...
if ( y * 2 = z / 2 ) ...
if ( x / 2 == *10 ) ...
if ( c == b ) ...
if ( a > x && b > y ) ...
```

Figura 4. 2. Ejemplos de expresiones lógicas bien constituidas.

Una expresión lógica se escribe utilizando una combinación de operandos, operadores relacionales y operadores lógicos (ver tema 2.5). En la Figura 4.2 se muestran las características de las expresiones lógicas correctamente constituidas. Cabe señalar que en las expresiones lógicas pueden utilizarse tanto constantes como variables, siempre y cuando estas últimas hayan sido previamente declaradas e inicializadas dentro del programa.

De acuerdo a las necesidades propias de la solución, hay tres situaciones en las que puede requerirse el uso de una condición.

Estas situaciones, y ejemplos para cada una de ellas, son:

1. Cuando se presenta una acción que sólo debe realizarse bajo una situación específica.
   - "Si el total de la compra de un cliente es mayor a mil pesos, aplicar un descuento de cinco por ciento".
2. Cuando se presentan dos acciones alternativas, mutuamente excluyentes, de las cuales sólo puede realizarse una de ellas.
   - "Si el promedio de goles por partido es mayor o igual a 2 escribir la palabra Alto, de lo contrario escribir Normal".
3. Cuando se presentan tres o más acciones alternativas.
   - "Dependiendo del número entero que representa la estación del año, mostrar la palabra correspondiente al nombre de la estación".

En la Figura 4.3 se muestran las implicaciones de cada uno de los tres casos y el diagrama de flujo correspondiente:

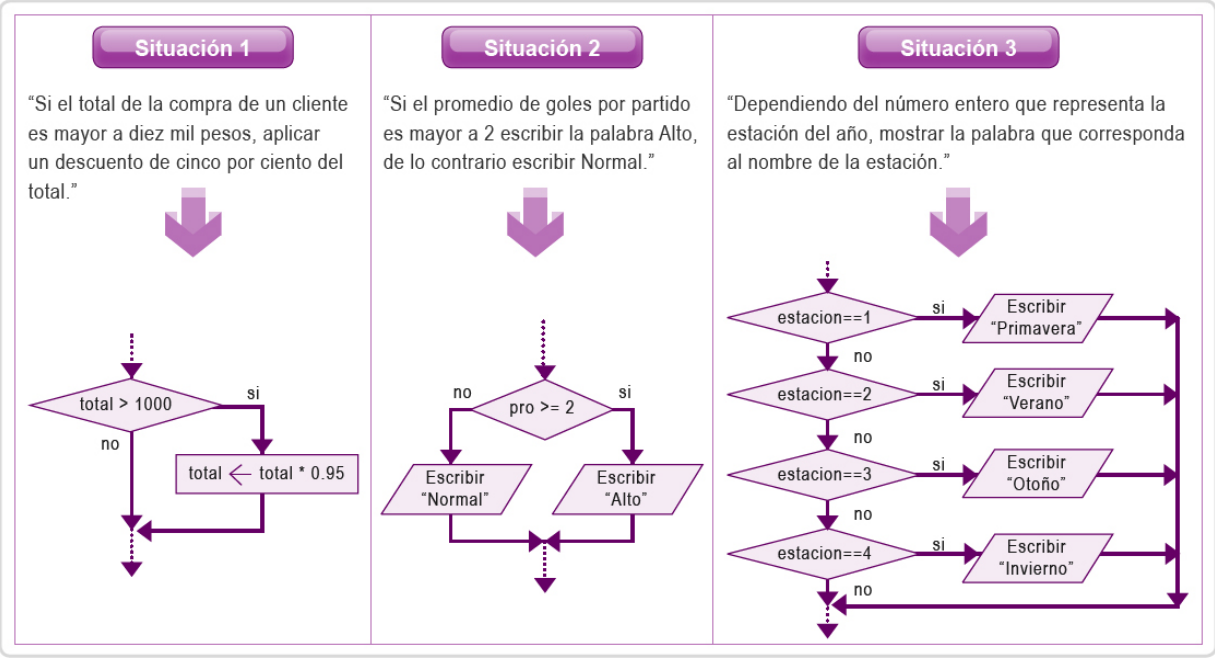


Figura 4. 3. Uso de condicionales e instrucciones de decisión.

De la Figura 4.3 puede inferirse mucho sobre las implicaciones de cada una de las situaciones mostradas. Durante el análisis del problema es muy importante que el lector identifique a cuál de los tres tipos de situaciones se enfrenta para determinar la estructura que debe utilizar. Use la Figura 4.4 para identificar la sintaxis de las estructuras condicionales.

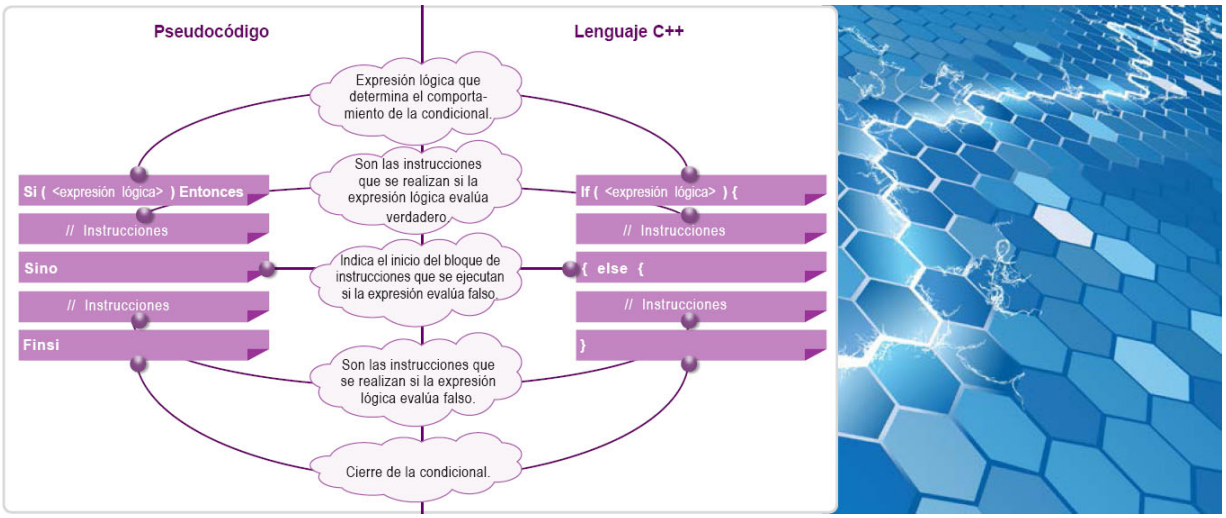


Figura 4. 4. Pseudocódigo y código fuente de cada caso en C++.

Con base esas reglas de sintaxis es posible escribir correctamente cada uno de los tres casos mencionados. En la Figura 4.5 se muestran las expresiones en pseudocódigo y en C++.

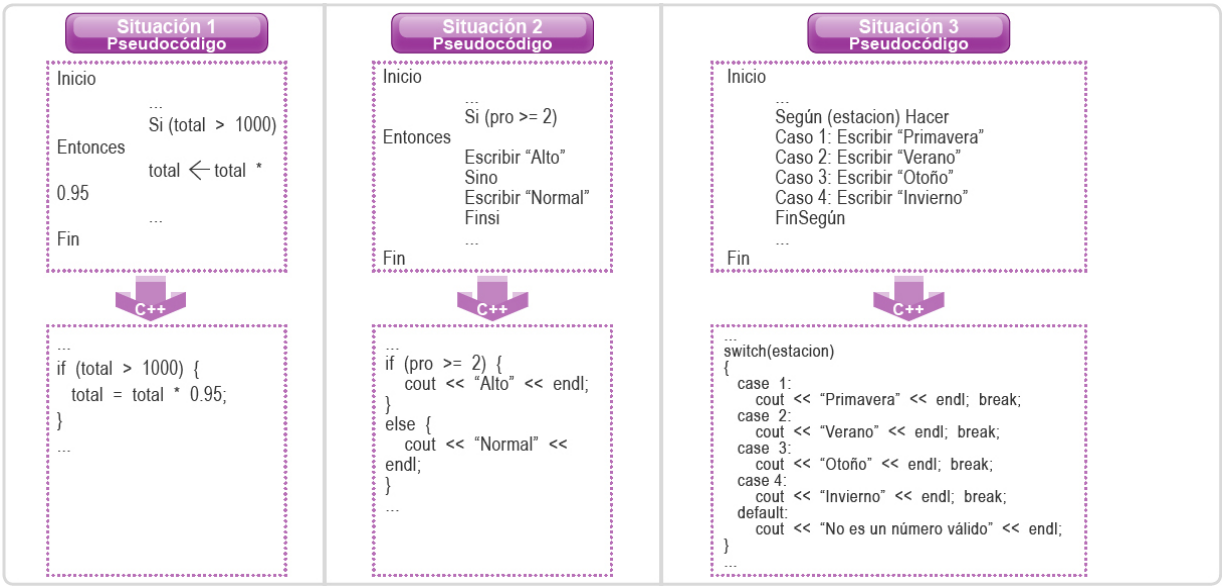


Figura 4. 5. Pseudocódigo y código fuente de cada caso en C++.

Note que en el caso de la instrucción switch las expresiones no son lógicas sino numéricas, esto se debe a que puede haber más de dos alternativas.

Una particularidad de la instrucción switch es que requiere de la instrucción *break* para salir del bloque de instrucciones correspondientes a un caso. De no emplear la instrucción break, el programa continuaría revisando las siguientes instrucciones dentro de switch de manera secuencial hasta el final.

Por último, el caso denominado *default* es el que se realiza cuando ninguno de los casos previos se cumple. Se considera una buena práctica definir el caso *default* debido a que ofrece una alternativa última al flujo del programa y previene comportamientos inesperados del mismo.

A continuación se abordan ejemplos del uso de condicionales en el contexto de la solución a problemas específicos. La Figura 4.6 identifica el criterio empleado en el diseño del algoritmo y el uso de condicionales dentro de este.

Problema

Recientemente ha llegado a la empresa una nueva máquina embotelladora de refrescos, el contenedor principal de la máquina tiene forma cilíndrica. Se sabe que cada envase de refresco debe contener ***M*** mililitros. Se desea saber cuántos refrescos completos puede llenar la máquina de una sola vez, sin recargar el contenedor. Solo se tienen los datos del radio de la base y la altura del contenedor medidos en metros.

Diseño del algoritmo

Datos de entrada:
- El número de variaciones por segundo.

Datos de salida:
- El mensaje de alarma correspondiente: "Normal", "Alerta amarilla" o "Alerta roja".

Planteamiento del problema:
Dado el número de variaciones por segundo, si este es menor a 40 presentar "Normal", si es mayor o igual a 40 y menor a 80 presentar "Alerta amarilla"; finalmente si es mayor o igual a 80 mostrar "Alerta roja".

Diseño del algoritmo

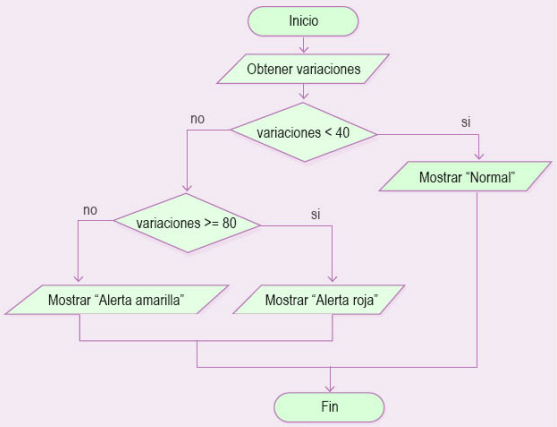
Consideraciones de diseño:

- Se tienen tres casos bien diferenciados.
- Se puede ir descartando un caso a la vez.
- Se puede iniciar con cualquiera de los dos rangos exteriores y descartar posteriormente los dos rangos restantes.

Diseño del algoritmo

```
Inicio
    Leer variaciones
    Si (variaciones < 40) Entonces
        Escribir "Normal"
    Sino
        Si (variaciones >= 80) Entonces
            Escribir "Alerta roja"
        Sino
            Escribir "Alerta amarilla"
        Finsi
    Finsi
Fin
```

Diseño del algoritmo



Elaboración del programa

```
#include<iostream>
using namespace std;

int main(){
     double variaciones;
     cout << "Indique número de variaciones por
segundo: ";
     cin >> variaciones;

     if(variaciones < 40){
          cout << "Normal" << endl;
     }else{
          if(variaciones >= 80){
               cout << "Alerta Roja" << endl;
          }else{
               cout << "Alerta Amarilla" << endl;
          }
     }

     return 0;
}
```

Elaboración del programa (opción 2)

```
#include<iostream>
using namespace std;

int main(){
     double variaciones;
     cout << "Indique número de variaciones por segundo: ";
     cin >> variaciones;

     if(variaciones < 40){
          cout << "Normal" << endl;
     }else{
          if(variaciones >= 80){
               cout << "Alerta Roja" << endl;
          }else{
               cout << "Alerta Amarilla" << endl;
          }
     }

     return 0;
}
```

Verificación del programa

Salida a pantalla

```
Indique el número de variaciones por segundo:
50
Alerta Amarilla

...

Indique el número de variaciones por segundo:
10
Normal

...

Indique el número de variaciones por segundo:
120
Alerta Roja
```

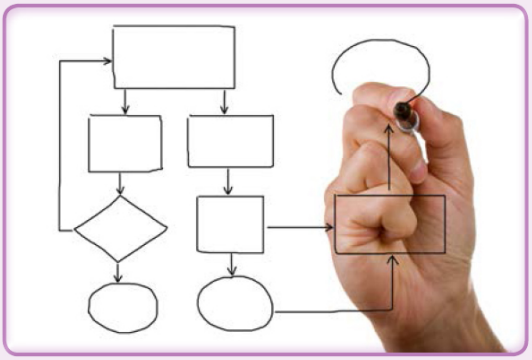


Figura 4. 6. Solución al problema del sensor sísmico.

Cabe hacer énfasis en que si la **expresión lógica** es evaluada como verdadera, se realizan las instrucciones ubicadas enseguida del if, dichas instrucciones conforman a su vez un bloque delimitado por una llave de apertura y una de cierre.

Sólo hay un caso en que las llaves pueden omitirse en un bloque, cuando el bloque consta de sólo una instrucción.

Por ejemplo, observe una muestra del programa correspondiente al problema recién analizado, sin el uso de las llaves.

Solución al problema de las variaciones

```
#include<iostream>
using namespace std;

int main(){
        double variaciones;
        cout << "Indique número de variaciones por segundo: ";
        cin >> variaciones;

        if(variaciones < 40) cout << "Normal" << endl;
        else if(variaciones >= 80) cout << "Alerta Roja" << endl;
                else cout << "Alerta Amarilla" << endl;

        return 0;
}
```

Figura 4. 7. Solución alternativa al problema del sensor sísmico.

Cabe señalar que las soluciones que se presentan son sólo dos de las muchas posibilidades para resolver el problema del sensor sísmico.

Analice la Figura 4.8 para identificar algunas de las soluciones alternativas para el mismo problema.

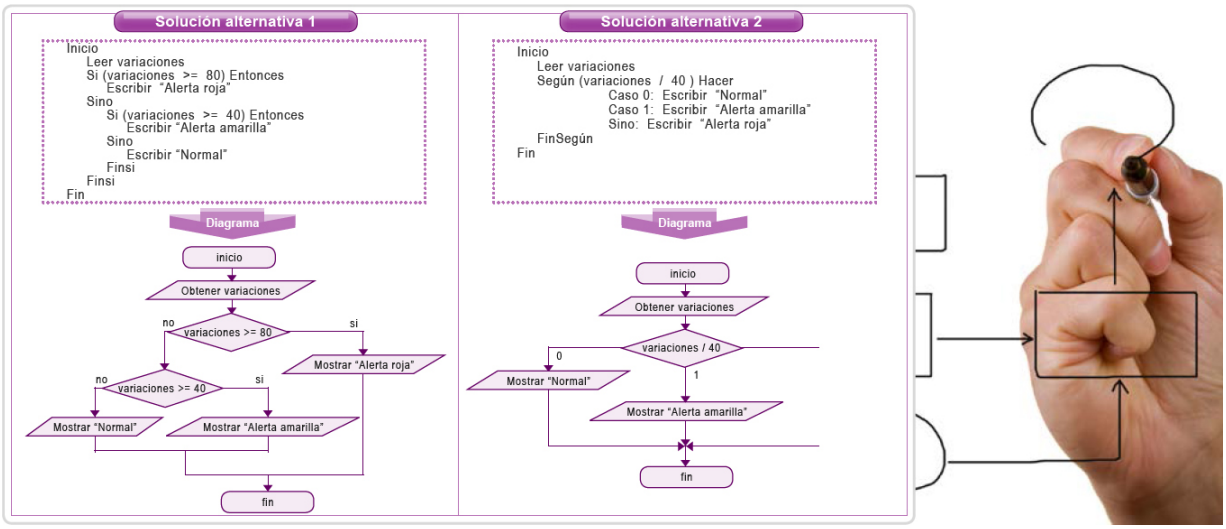
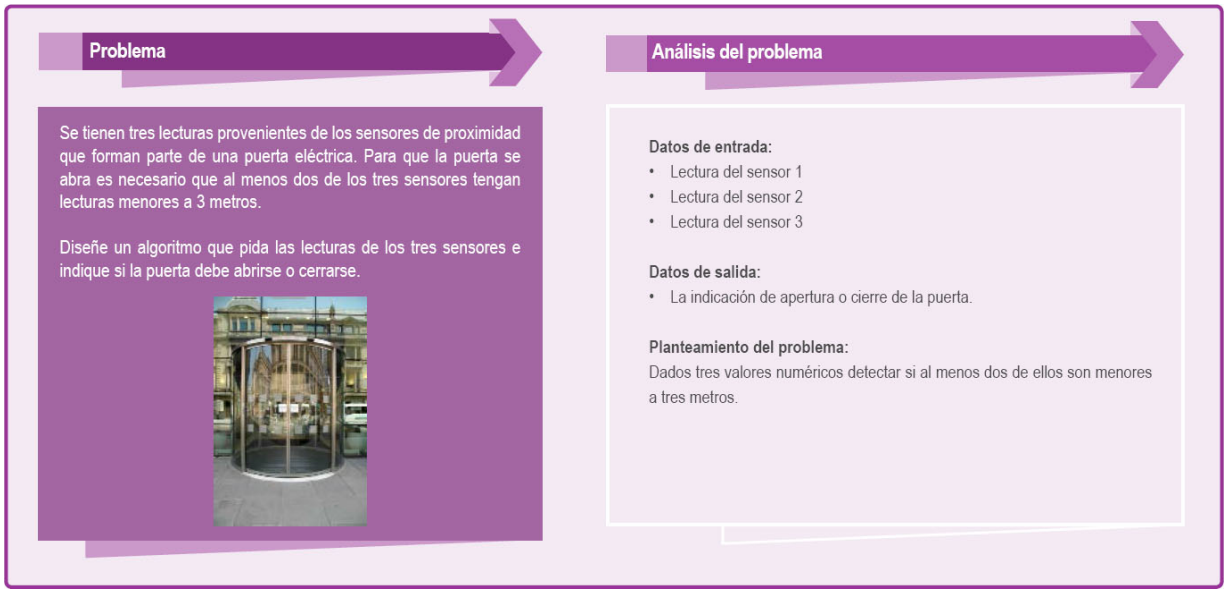


Figura 4. 8. Soluciones alternativas al problema del sensor sísmico.

En la Figura 4.9 se analiza otro ejemplo del uso de las instrucciones condicionales y de selección.



Diseño del algoritmo

Consideraciones de diseño:

- Se pueden usar combinaciones para identificar dos medidas menores a tres metros.
  - Verificar sensor 1 y sensor 2
  - Verificar sensor 1 y sensor 3
  - Verificar sensor 2 y sensor 3
- Puede también contarse el número de sensores que están activados.
  - Se requeriría una variable adicional para llevar la cuenta.
  - Una vez contados se verificaría el número de sensores que cumplieron la condición.
- Si los tres están activados detonaría las tres condiciones usando las combinaciones, sería necesario usar un else después de cada combinación.

Diseño del algoritmo

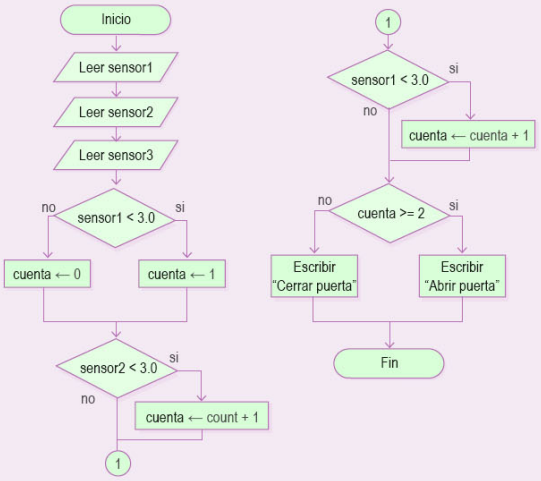
```
Algoritmo sensores
Variables
   sensor1, sensor2, sensor3, cuenta
Inicio
   Leer sensor1
   Leer sensor2
   Leer sensor3

   cuenta ← contarPuertas(sensor1,
sensor2, sensor3)

   Si (cuenta >= 2) Entonces
      Escribir "Abrir puerta"
   Sino
      Escribir "Cerrar puerta"
   Finsi
Fin
```

```
Función  contarPuertas(  sensor1,
sensor2, sensor3)
Variables
   contador
Inicio
   Si (sensor1 < 3.0) Entonces
      contador ← 1
   Sino   contador← 0
   Finsi
   Si (sensor2 < 3.0) Entonces
      contador ← contador + 1
   Finsi
   Si (sensor3 < 3.0) Entonces
      contador ← contador + 1
   Finsi
   Devolver contador
Fin
```

Diseño del algoritmo



Elaboración del programa

```
#include <iostream>
using namespace std;

int contarPuertas(double, double, double);
int main(){
     double sensor1, sensor2, sensor3;
     int cuenta;

     cout << "Indique valor del sensor 1: "; cin >> sensor1;
     cout << "Indique valor del sensor 2: "; cin >> sensor2;
     cout << "Indique valor del sensor 3: "; cin >> sensor3;

     cuenta = contarPuertas(sensor1, sensor2, sensor3);

     if(cuenta >= 2) cout << "Abrir puerta" << endl;
     else cout << "Cerrar puerta" << endl;

     return 0;
}
```

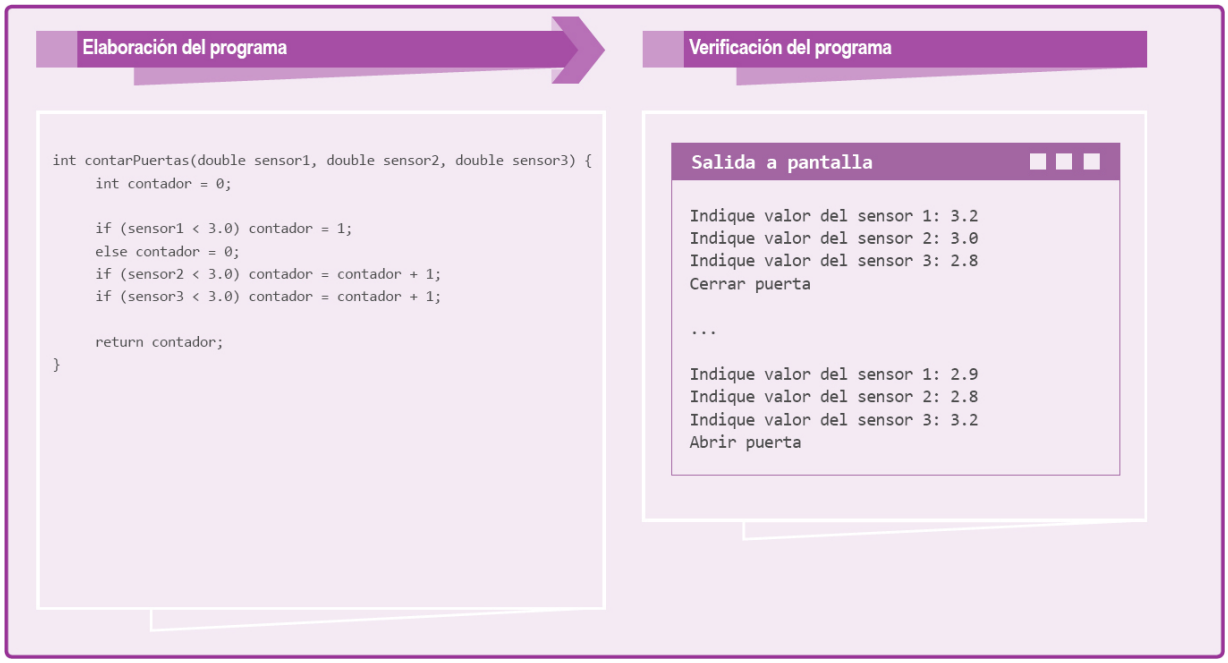


Figura 4. 9. Solución al problema de los sensores de una puerta automática.

De la misma forma que el problema de la alarma sísmica, para el problema de la puerta automática hay varias posibles soluciones. Observe en la Figura 4.10 algunas alternativas de solución.

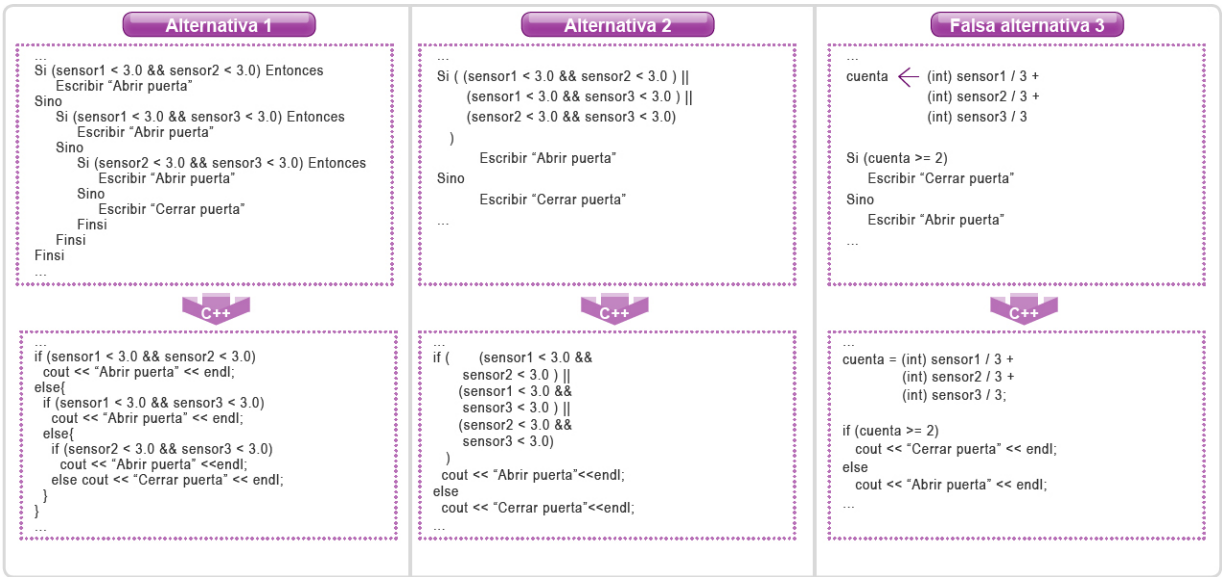


Figura 4. 10. Soluciones alternativas problema de la puerta automática.

En la Figura 4.11 se analiza un problema más, ahora relacionado con la estructura de decisión switch.

Problema

Elabore un programa que dada una fecha específica, indique el nombre del día de la semana correspondiente a la fecha en cuestión.

Use las fórmulas conocidas para determinar el día de la semana de una fecha dada: día/mes/año dada. El valor resultante d es el día de la semana: 0 significa domingo, 1 lunes y así sucesivamente.

$$a = \frac{14 - month}{12}$$

$$y = año - a$$

$$m = mes + 12a - 2$$

$$d = (día + y + \frac{y}{4} - \frac{y}{100} + \frac{y}{400} + \frac{31m}{12}) \bmod 7$$

Análisis del problema

Datos de entrada:

- el día, mes y año correspondiente a la fecha.

Datos de salida:

- El nombre del día de la semana correspondiente.

Planteamiento del problema:

Dados el día, mes y año de una fecha específica, determinar el nombre del día de la semana correspondiente.

Diseño del algoritmo

```
Algoritmo día_de_la_semana
Variables
    dd, mm, aaaa
Inicio
    Leer dd
    Leer mm
    Leer aaaa

    escribirDia(dd, mm, aaaa)
Fin
```

Diseño del algoritmo

```
Función escribirDia (aa, mm, dddd)
Variables
    a, y, m, d
Inicio
    a ← (12 – mm) / 12
    y ← aaaa – a
    m ← mm + 12*a – 2
    d ← (dd + y + y/4 – y/100 + y/400 + 31*m/12) mod 7

    Según (d) Hacer
        Caso 0: Escribir "domingo"
        Caso 1: Escribir "lunes"
        Caso 2: Escribir "martes"
        Caso 3: Escribir "miércoles"
        Caso 4: Escribir "jueves"
        Caso 5: Escribir "viernes"
        Caso 6: Escribir "sábado"
    FinSegún
Fin
```

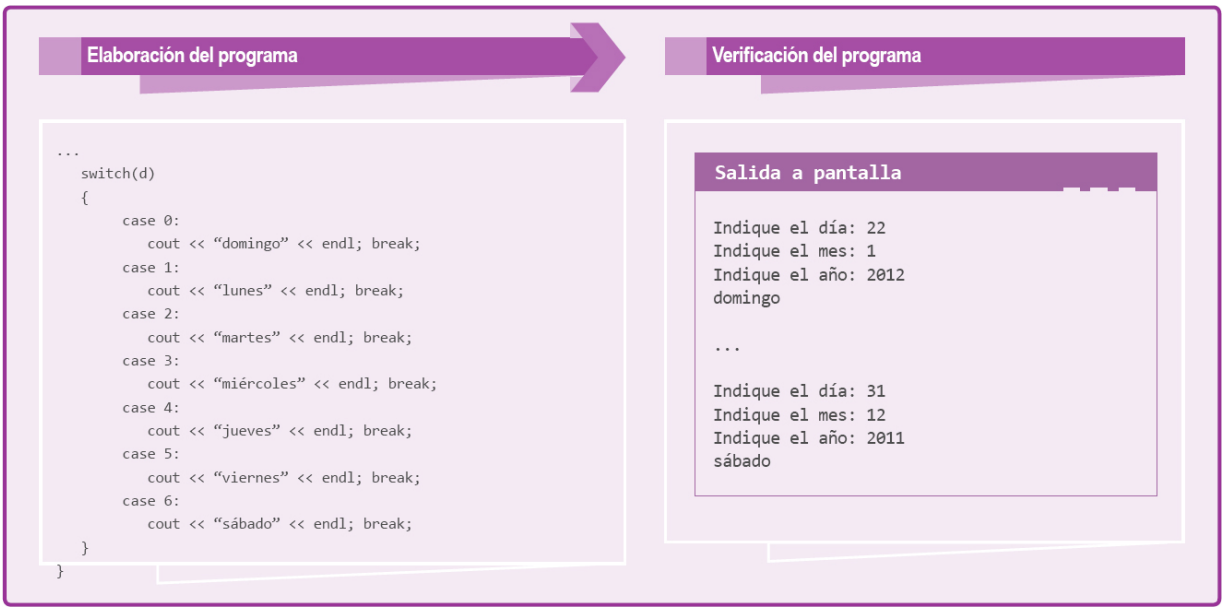


Figura 4. 11. Solución al problema del nombre del día de la semana.

La solución particular que una persona diseña depende en gran medida del análisis que realiza y del enfoque que la da. A lo largo de este eBook, el lector encontrará señalamientos relacionados con soluciones alternativas a los problemas que se presentan, además de observaciones relativas a la eficiencia en tiempo y espacio de las soluciones propuestas.

4.2 Expresiones lógicas simples y compuestas

En los ejemplos que se mostraron en este capítulo se abordaron expresiones lógicas simples que utilizan operadores relacionales y también expresiones lógicas compuestas. Estas últimas son producto de expresiones lógicas simples combinadas mediante el uso de operadores lógicos. Estos operadores son And, Or y Not, se escriben en C++ usando los símbolos &&, || y ! respectivamente (ver tema 2.5). Estos operadores permiten combinar los valores de verdad que resultan de expresiones lógicas simples para escribir expresiones compuestas.

Para determinar el valor de verdad resultante, basta con recordar que el operador And sólo devuelve verdadero

cuando ambas expresiones son verdaderas, mientras que el operador Or devuelve verdadero cuando al menos una de las expresiones es verdadera.

Observe algunos ejemplos de situaciones en las que es necesario escribir expresiones lógicas compuestas. Al interactuar con la Figura 4.12 puede poner en práctica los conocimientos de expresiones lógicas compuestas.

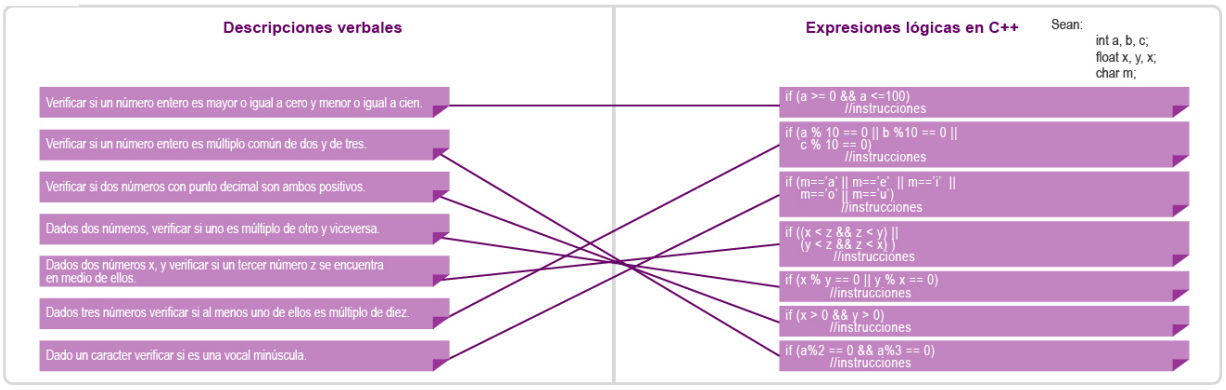


Figura 4. 12. Expresiones lógicas compuestas.

En algunos casos puede ser difícil escribir correctamente una expresión lógica, sobre todo cuando en las restricciones intervienen varias variables. Se sugiere al lector verbalizar la restricción y poner atención a la oración construida para identificar cuáles son los operadores lógicos que deben utilizarse. Palabras como *ambos* o *todos* denotan que se trata de un operador *And;* mientras que la frase *al menos uno* sugiere el uso del operador *Or.*

Las condicionales no son solo un mecanismo para dividir el flujo de un algoritmo. Una expresión lógica bien escrita, puede hacer un programa más eficiente, más claro o más fácil de entender y modificar. En general, las condicionales son uno de los elementos más importantes de la programación estructurada. Por medio de ellas se expresa una gran variedad de restricciones que los problemas específicos abordan. Con el uso de las condicionales y las estructuras de decisión, el lector comienza a tener en sus manos una variedad, cada vez mayor, de herramientas para el diseño de soluciones. Es importante que el lector se familiarice con estas

expresiones, ya que dominarlas correctamente es muy importante para continuar con los temas del curso.

4.3 Repeticiones for, while

La automatización de procedimientos es una de las necesidades más comunes de las personas y de las empresas. La posibilidad de programar una computadora para que realizara las tareas cotidianas repetitivas fue uno de los detonadores iniciales de la era de la computación. En la actualidad, las tareas que se llevan a cabo con ayuda de una computadora son mucho más complejas, no obstante, el procesamiento sigue fundamentándose en las mismas estructuras de control de ejecución.

A las estructuras de control que permiten realizar varias veces un segmento de código se les denomina *ciclos.*

Un **ciclo** tiene siempre asociada una expresión lógica; esta expresión sirve para determinar si el bloque de instrucciones asociadas al ciclo debe ejecutarse. Si la expresión lógica evalúa *verdadero,* las instrucciones dentro del bloque del ciclo se llevan a cabo.

El lector habrá notado que la información del párrafo anterior corresponde fielmente a lo que sería una instrucción *if,* sin embargo, justo al finalizar el bloque se diferencian. Mientras una estructura condicional cedería el control a las instrucciones posteriores, en el

caso del ciclo, el programa regresa automáticamente a verificar la expresión lógica. Este salto *hacia atrás* se realiza de manera automática e incondicional y establece la drástica diferencia entre una expresión condicional y un ciclo. En la Figura 4.13 se explica el concepto de ciclo y su similitud con la instrucción *if.*

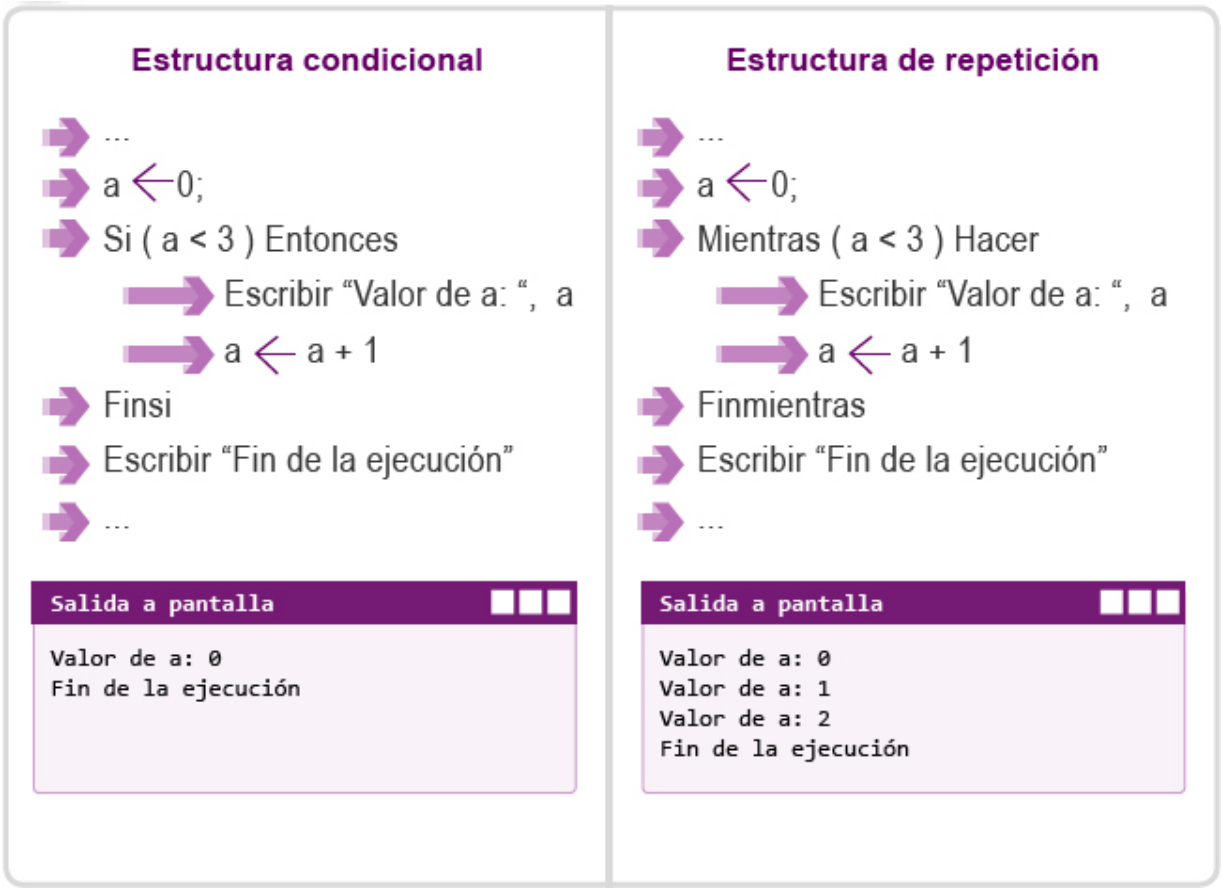


Figura 4. 13. Concepto de ciclo y su relación con las estructuras condicionales.

Para utilizar de manera correcta un ciclo, además de la expresión lógica, es necesario escribir una instrucción dentro del ciclo que modifique el valor de alguna de las variables involucradas en la expresión lógica. De no hacerlo, el valor de verdad de la expresión lógica no cambiaría, lo que ocasionaría que el ciclo continuara indefinidamente. En la Figura 4.14 se muestra la sintaxis del ciclo *while* y las partes que lo componen.

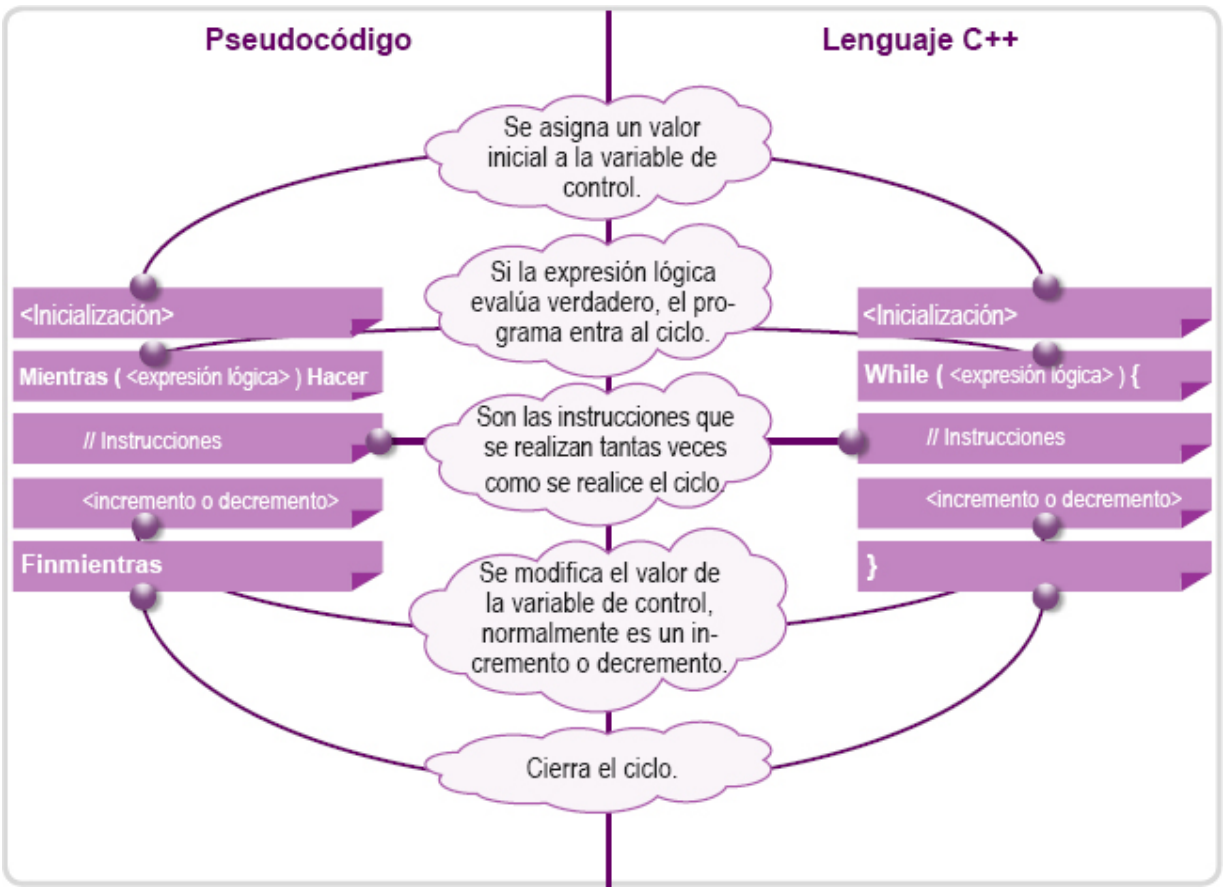


Figura 4. 14. Sintaxis de la estructura de control while.

El comportamiento de un ciclo depende básicamente de la combinación de tres elementos: la inicialización, la expresión lógica y la modificación de las variables involucradas en la expresión lógica. Para el lector es primordial saber, que en términos generales, un ciclo while podría no ejecutarse ni siquiera una vez; esto en caso de que la expresión lógica evalúe *falso* al intentar ingresar por primera vez al ciclo. En la Figura 4.15 el lector puede revisar un ejercicio para reforzar el conocimiento de los ciclos.

Código en vivo

```
#include <iostream>

using namespace std;


int main() {

  int radio = 2;

  while (radio < 6 ){

    dibujarCirculo( 7 , 0 , radio );

    radio = radio + 1 ;

  }

}
```

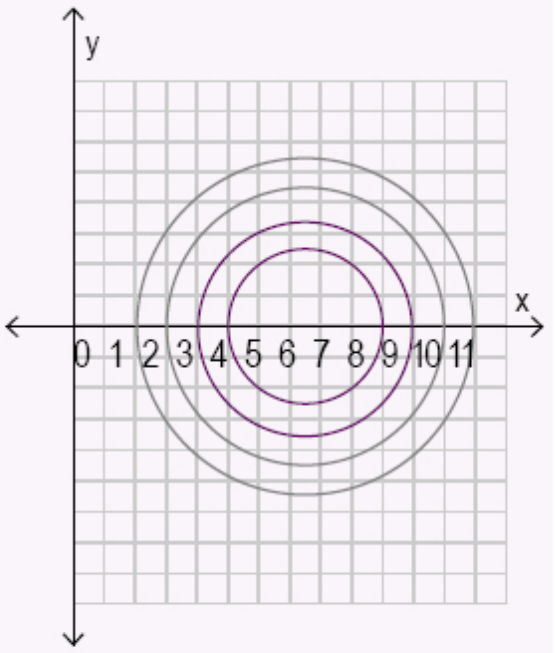
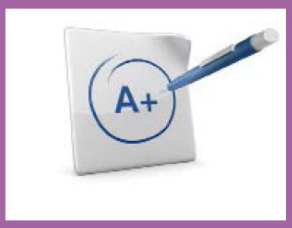


Figura 4. 15. Ejercicio para reforzar la comprensión de los ciclos.

Cabe resaltar que la expresión lógica reguladora del funcionamiento del ciclo debe escribirse considerando que al evaluar *verdadero* el ciclo continuará ejecutándose. Se menciona lo anterior ya que, con frecuencia, se logra visualizar la condición de salida. Cuando este sea el caso, basta con usar el operador negación *!* sobre la expresión completa para tener entonces la expresión lógica buscada. En la Figura 4.16 se observa el uso de ciclos para resolver un problema específico.

Problema

Elabore un programa que sea capaz de procesar las notas de un grupo de alumnos. El programa deberá solicitar el número de alumnos en el grupo y enseguida pedir por teclado la nota de cada uno en una escala de 0 a 100. Una vez capturadas deberá mostrar el promedio resultante, la nota más alta y la más baja.



Análisis del problema

Datos de entrada:
- El número de alumnos a procesar
- La nota de cada alumno

Datos de salida:
- El promedio del grupo.
- La nota más alta
- La nota más baja.

Planteamiento del problema:
Dadas N calificaciones, calcular el promedio, la nota más alta y la más baja.

Diseño del algoritmo

Consideraciones de diseño:

- Es necesaria una variable para ir almacenando la suma de las calificaciones.
- Al terminar el ciclo se puede calcular el promedio dividiendo entre el número total de calificaciones.

- Sólo se dispone de la última nota por lo que el cálculo del mayor y menor debe hacerse de manera acumulativa.
  - El menor puede iniciar en 100 de esta forma cualquier valor dentro del rango 0...100 provocará que se actualice la variable.
  - El mayor puede iniciar en 0 de esta forma cualquier valor dentro del rango 0...100 provocará que se actualice la variable.

Diseño del algoritmo

```
#include <iostream>
using namespace std;

int main(){
   int total, nota, suma, alumno, mayor, menor;
   double promedio;

   cout << "Indique el total de alumnos en el grupo: ";
   cin >> total;

   mayor = 0;
   menor = 100;
   suma = 0;
   alumno = 1;
...
```

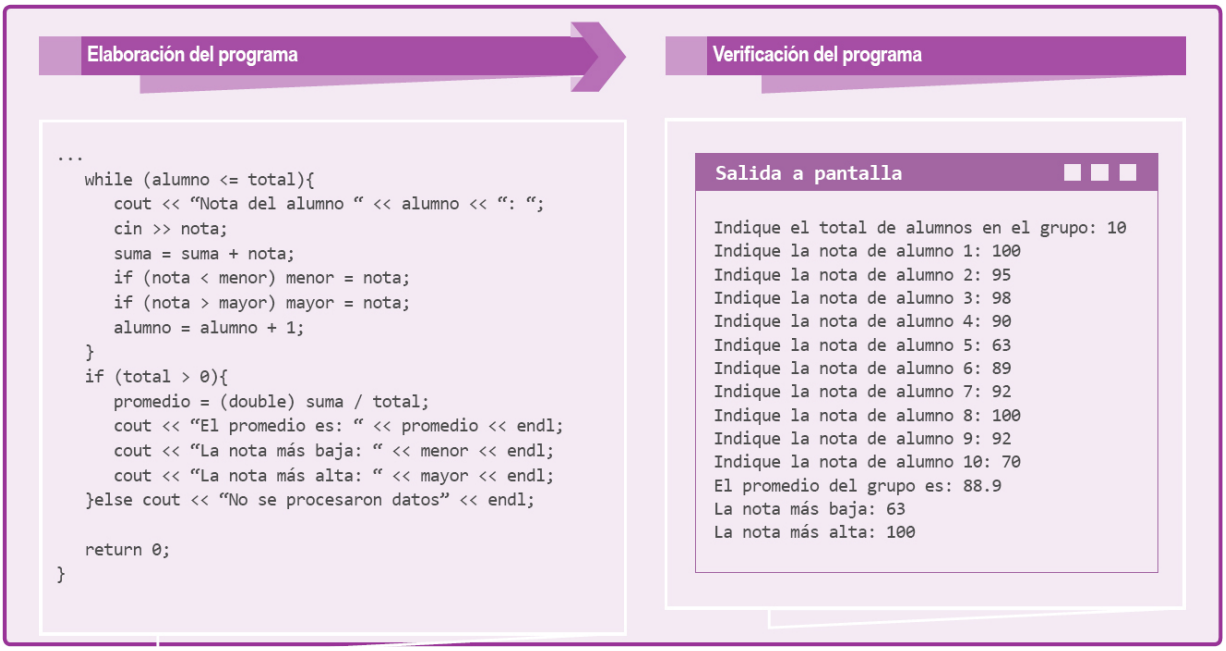


Figura 4. 16. Solución al problema de las calificaciones.

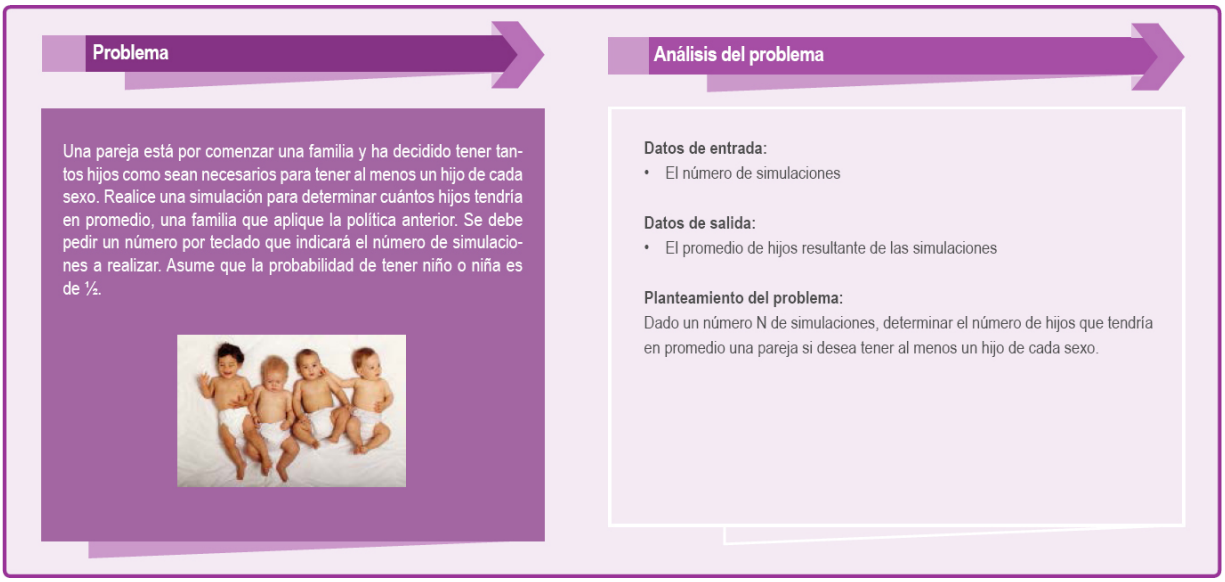
Del problema anterior se desprenden cuatro observaciones importantes:

1. La variable *nota* es reutilizada varias veces para almacenar la calificación en turno y, cada vez que es reutilizada, el valor contenido es sobrescrito.
2. La variable *suma* inicia en cero y va acumulando los valores obtenidos en *nota* a cada paso del ciclo. A las variables con este comportamiento se les denomina **acumuladores**.
3. La variable *alumno* controla el ciclo al llevar la cuenta del alumno que se está procesando, es también la que interviene en la expresión lógica. A las variables con este comportamiento se les denomina **contadores**.
4. Las variables *mayor* y *menor* sólo almacenan un valor, y actualizan en cuanto identifican, por medio de una condición, que la calificación en turno es mayor o menor que la almacenada hasta ese momento.

Conviene familiarizarse con los comportamientos antes descritos debido a que resultan de gran utilidad en la resolución de una amplia variedad de problemas. Saber cuándo una variable debe acumular, contar incondicionalmente o contar dependiendo de una condición, es uno de los puntos críticos para comprender el funcionamiento de los ciclos y aprovecharlos al máximo.

Piense por un momento en el número de instrucciones que habría tomado diseñar la solución al problema de calificaciones sin el uso de ciclos. Por supuesto, podría haberse colocado una condicional seguida de otra, pero dicha solución ocuparía una cantidad mucho mayor de líneas de código, sin mencionar que se tendría que conocer de antemano en número de notas por procesar.

Los ciclos, al igual que las estructuras condicionales, pueden utilizarse en forma anidada, es decir, puede colocarse a un ciclo dentro de otro ciclo. Observe la Figura 4.17 y analice con detenimiento la solución a un problema de simulación.



Diseño del algoritmo

Consideraciones de diseño:

- Una simulación consiste en generar aleatoriamente valores binarios (masculino/femenino) hasta que exista al menos uno de cada sexo.
- Se requieren dos variables una para contar los hijos de cada sexo.
- Al terminar una simulación debe reiniciar los contadores y realizar de nuevo otra simulación.
- Se requiere un ciclos anidado dentro de otro para hacer N experimentos.

Diseño del algoritmo

```
#include <iostream>
#include <cstdlib>

using namespace std;

int simulaPareja();

int main(){
   int simula, N, suma;
   double promedio;

   cout << "Número de simulaulaciones: "; cin >> N;
   srand(time(0));
   simula = 1;
   suma = 0;

...
```

Elaboración del programa

```
...

   while (simula <= N) {
      suma = suma + simulaPareja();
      simula = simula + 1;
   }
   promedio = (double) suma / N;
   cout << "En promedio una pareja tendría " <<
promedio
        << " bebés para tener uno de cada sexo"
<< endl;

   return 0;
}
```

Elaboración del programa

```
int simulaPareja(){
    int hombres, mujeres, hijo;

    hombres = mujeres = 0;
    while(hombres < 1 || mujeres < 1){
        hijo = rand() % 2;
        if ( hijo == 0 ) hombres = hombres + 1;
        else mujeres = mujeres + 1;
    };

    return hombres + mujeres;
}
```

Verificación del programa

Salida a pantalla

```
Número de simulaulaciones: 10
Simulación 1 -> Hombres: 1, Mujeres: 3
Simulación 2 -> Hombres: 2, Mujeres: 1
Simulación 3 -> Hombres: 2, Mujeres: 1
Simulación 4 -> Hombres: 1, Mujeres: 3
Simulación 5 -> Hombres: 1, Mujeres: 1
Simulación 6 -> Hombres: 4, Mujeres: 1
Simulación 7 -> Hombres: 2, Mujeres: 1
Simulación 8 -> Hombres: 1, Mujeres: 1
Simulación 9 -> Hombres: 5, Mujeres: 1
Simulación 10 -> Hombres: 4, Mujeres: 1
En promedio una pareja tendría 3.7 bebés para
tener uno de cada sexo
```

Figura 4. 17. Solución al problema de los bebés.

En el problema anterior se observa el enorme potencial derivado del uso y desarrollo de herramientas computacionales en simulación, análisis y predicción para la toma de decisiones. De manera similar al caso de los bebés, es posible explorar diversos fenómenos, naturales o no, en distintas áreas del conocimiento.

Al escribir programas que utilizan ciclos, existe una alternativa para la estructura while: se trata de la estructura de repetición *for.* Esta estructura tiene el mismo comportamiento que while, la diferencia reside en la sintaxis. En la Figura 4.18 se muestra la sintaxis de la estructura de repetición for y la correspondencia de cada una de sus partes con la estructura while.

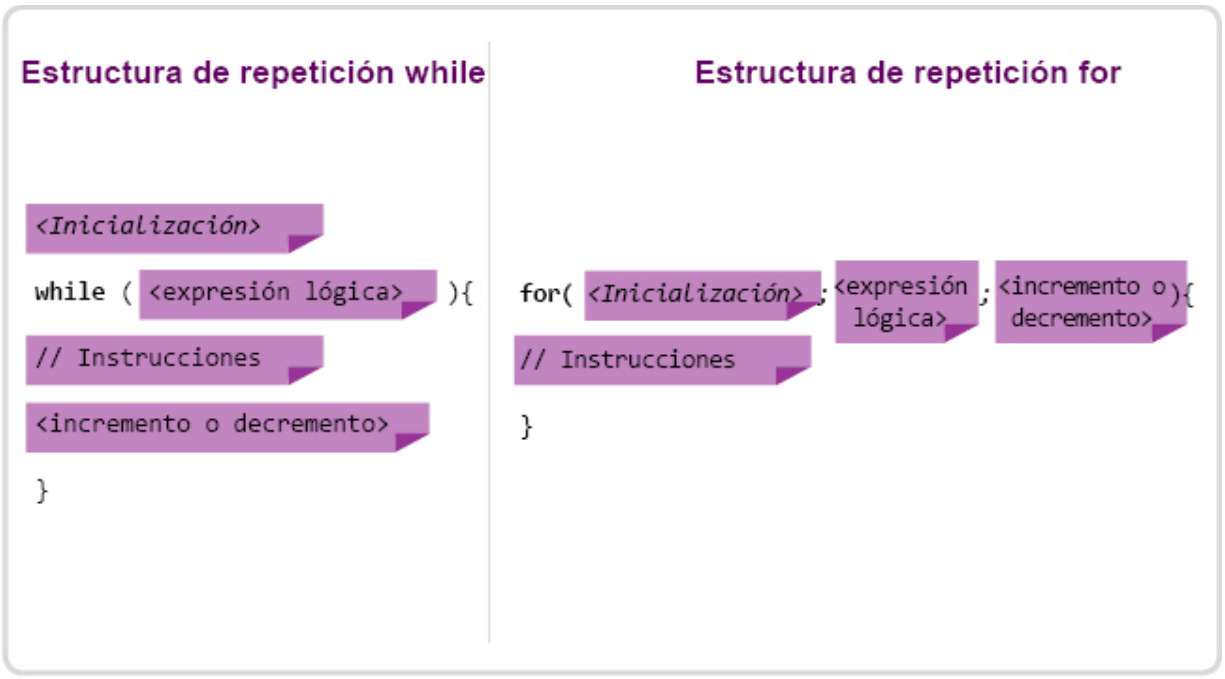


Figura 4. 18. Sintaxis de la estructura de repetición *for*.

El aspecto más importante del uso de la instrucción for radica el entendimiento del orden de ejecución implícito: la inicialización se realiza sólo una vez, la expresión lógica se evalúa siempre a la entrada del ciclo y la instrucción de actualización (incremento o decremento) se lleva a cabo siempre al final de cada ciclo. En la Figura 4.19 podrá poner a prueba sus conocimientos sobre las estructuras de repetición usando la sintaxis definida para la estructura for.

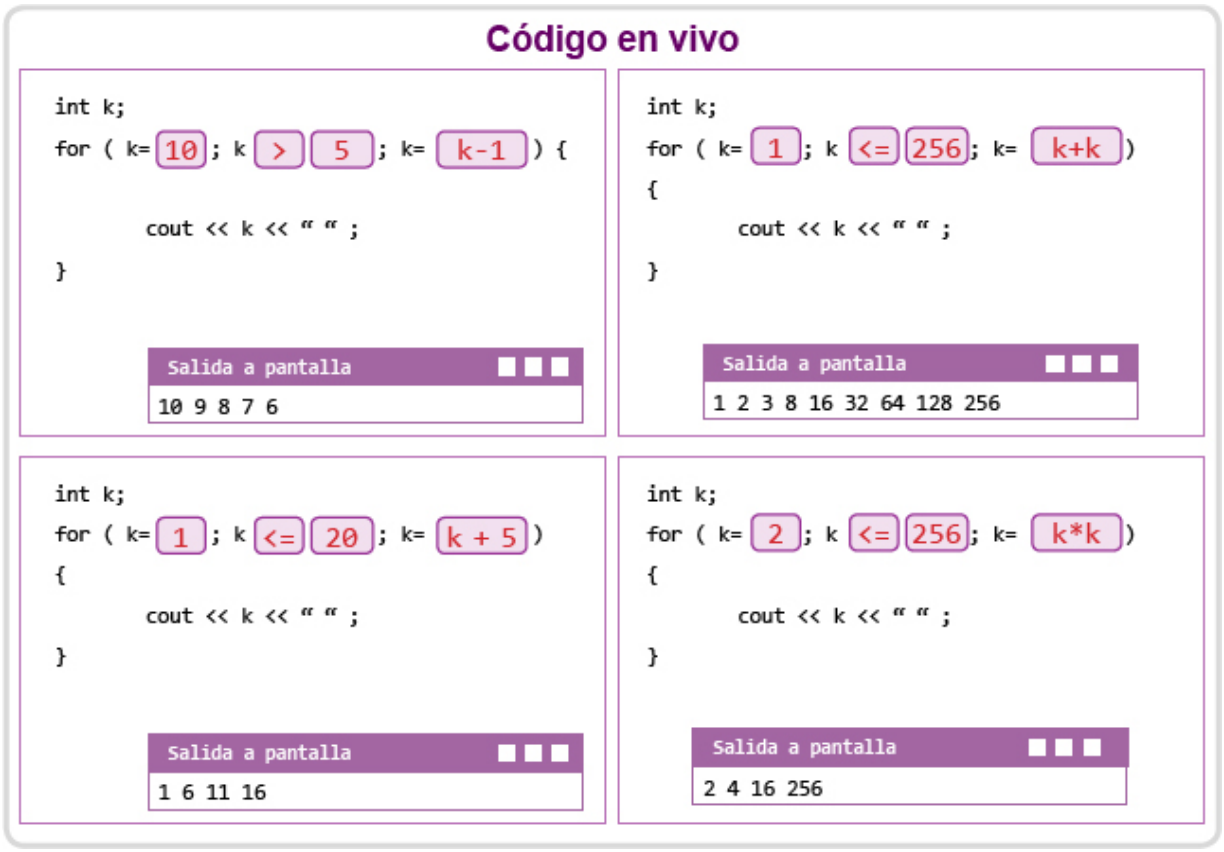


Figura 4. 19. Ejemplos de uso de la estructura de repetición *for*.

En términos generales, un ciclo que utiliza la instrucción while siempre puede escribirse usando una instrucción for y viceversa; en términos prácticos son equivalentes. La única diferencia radica en lo compacto de la notación. Mientras un ciclo escrito con while requiere de una instrucción por cada parte que lo compone, en los ciclos for la inicialización, expresión lógica y modificación de la variable se escriben en una sola instrucción.

4.4 Repeticiones *do-while*

Una vez que se han visto las estructuras de control while y for, toca el turno a la estructura *do-while.* Esta estructura es similar a las anteriores, ya que también se utiliza para realizar varias veces un conjunto definido de instrucciones.

La diferencia principal del do-while con respecto de los ciclos while y for es que un ciclo do-while siempre realiza, por lo menos una vez, el conjunto de instrucciones.

En un ciclo do-while la expresión lógica se evalúa al final del bloque. Si la expresión evalúa verdadero, entonces la ejecución del programa regresa al inicio del ciclo do, de lo contrario termina la ejecución del ciclo. En la Figura 4.20 se muestra la sintaxis del ciclo do-while y su funcionamiento.

Como se mencionó anteriormente un ciclo do-while se utiliza cuando las instrucciones dentro del bloque deben ser ejecutadas, por lo menos, en una ocasión. Esto depende del problema que se presente.

Por ejemplo: Considere el caso en que debe validarse una nota en un rango de 0 a 100. El segmento de código quedaría de la siguiente manera:

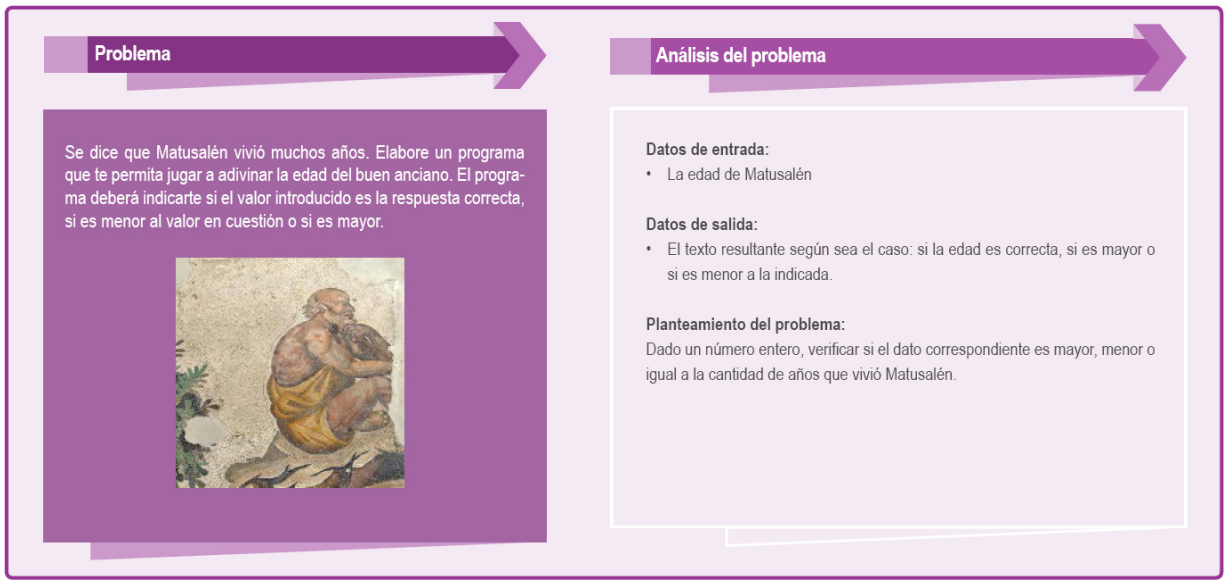
Figura 4. 20. Sintaxis de la estructura do-while.

```
...
do{
```

```
    cout << “Indique una nota (0-100): “;
    cin >> nota;
}while(nota < 0 || nota > 100);
...
```

Antes de verificar si la nota es válida o no, debe solicitarse por lo menos una vez. En caso de que el rango sea incorrecto la expresión lógica evaluará verdadero y el programa regresará automáticamente a la línea donde está la instrucción do. Esto último lo hará una y otra vez hasta el usuario ingrese un valor válido. En ese momento la expresión lógica evaluará falso y el programa continuará su ejecución.

En la Figura 4.21, se revisa un ejemplo de la aplicación del ciclo do-while en la solución a un problema específico.



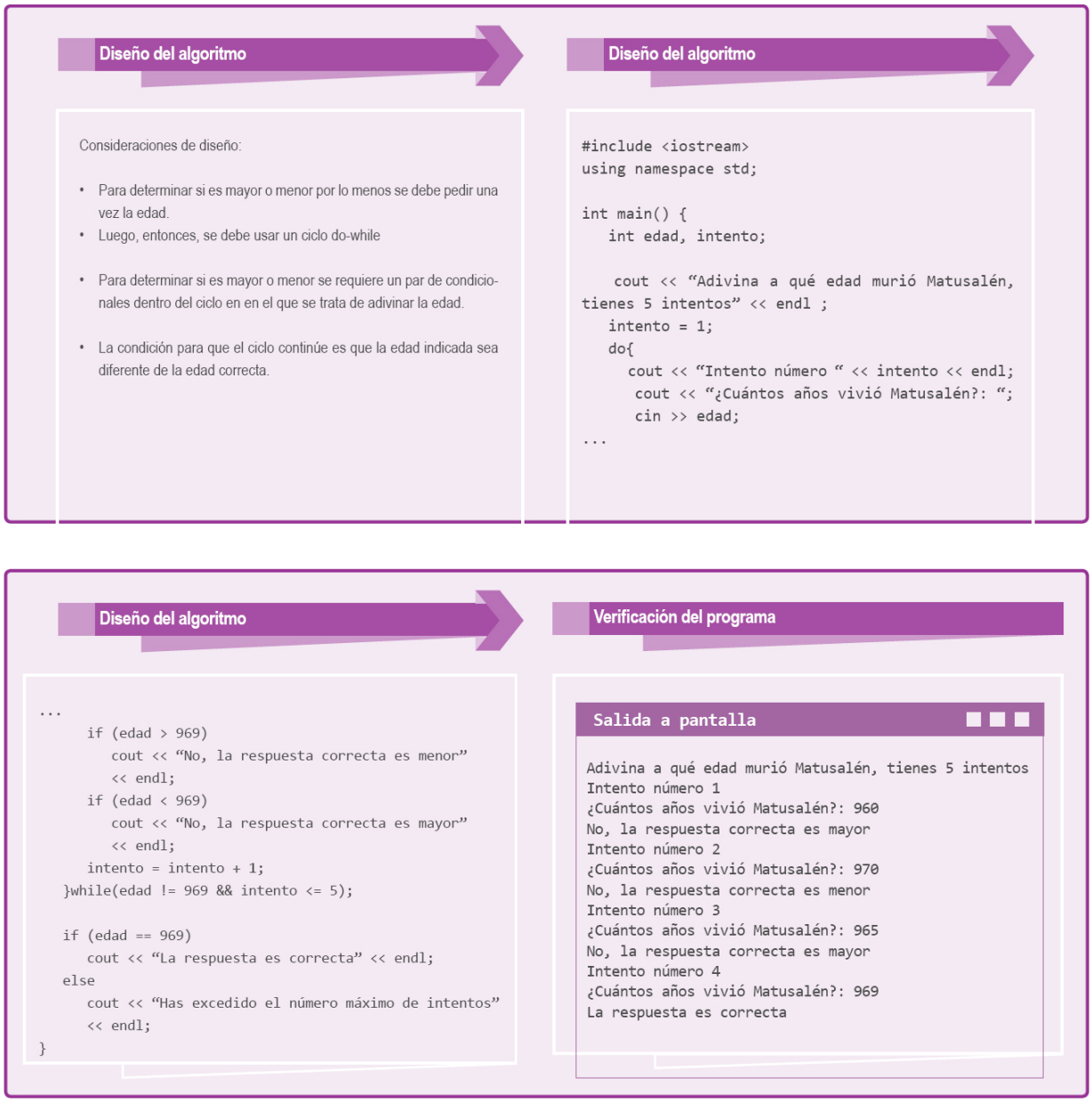


Figura 4. 21. Solución al problema de Matusalén.

En el problema anterior, es claro que la estimación de la edad debe introducirse por lo menos una vez para que el programa ofrezca alguna retroalimentación. De ahí que aplicar un ciclo do-while sea la manera más natural de abordarlo.

4.5 Criterios para el uso de ciclos

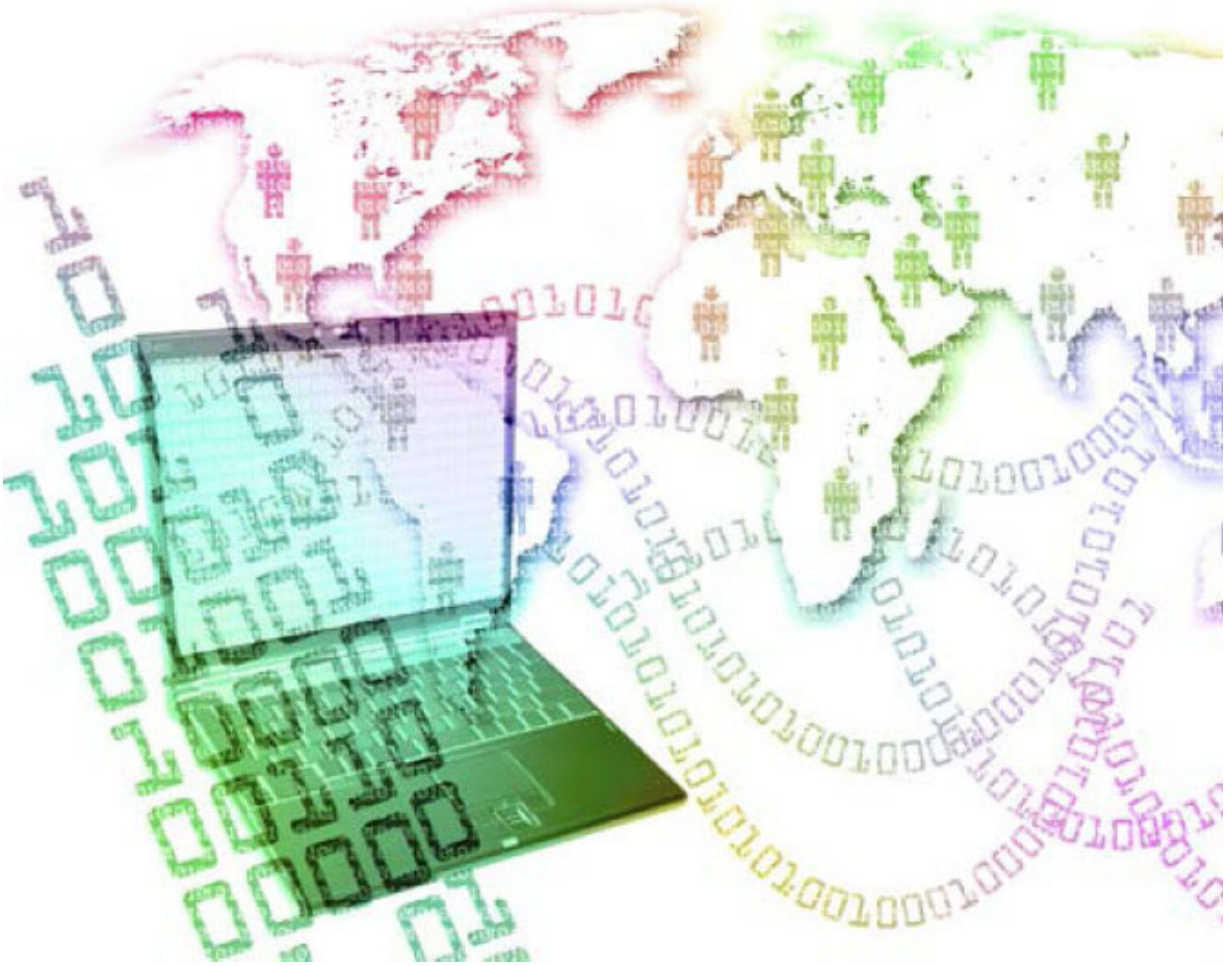
Es importante considerar tres criterios al diseñar aplicaciones que utilicen ciclos:

- El primero consiste en determinar si el conjunto de instrucciones dentro del ciclo se ejecutará por lo menos una vez; de ser así se debe optar por un ciclo do-while. En caso contrario, es preferible un ciclo while o un ciclo for, ya que estos dos últimos son equivalentes.

- El segundo criterio consiste en reconocer claramente los factores clave para escribir la expresión lógica que requiere el ciclo. Se sugiere identificar las relaciones que deben cumplirse entre variables y constantes con el objetivo de establecer las condiciones necesarias para que el ciclo continúe ejecutándose. Cabe recalcar que si la expresión evalúa verdadero el ciclo continúa. Incluso, si por alguna razón se formula la expresión lógica que identifica el caso en el que el ciclo debe terminar, puede usarse el operador lógico negación (!) para invertir el valor de verdad de dicha expresión y así utilizarla como la expresión lógica para el ciclo.

- El tercer criterio consiste en identificar, independientemente del ciclo utilizado, que la condición escrita arroje como resultado (tarde o temprano) un valor de verdad falso, ya que de lo contrario el programa podría continuar ejecutándose indefinidamente.

Si se vigila la correcta aplicación de los tres criterios descritos, será posible elaborar programas muy compactos, funcionales y más fáciles de modificar que los que hacen uso innecesario de muchas condicionales.



4.6 Piense en grande: minería de datos, procesar terabytes de información

Como se ha visto en este capítulo, los ciclos proporcionan un mecanismo para realizar tareas repetitivas sobre grandes volúmenes de datos. En la actualidad se estima que diariamente se genera un **terabyte** de información por lo que el procesamiento oportuno esta es un aspecto clave para las empresas. Ocultos en este mar de información están patrones, fenómenos, relaciones y causalidades, datos que, al analizarse, representan elementos de alto valor agregado para las empresas y las organizaciones y que incluso pueden proporcionar ventajas significativas con respecto a los competidores y pueden mejorar la capacidad de toma de decisiones y de análisis estratégico.

Al proceso que se lleva a cabo para extraer conocimiento valioso a partir de grandes volúmenes de datos se le denomina minería de datos. En el video que está en la barra lateral se muestra un panorama general y un caso práctico de análisis predictivo.

En esta área no solamente la programación es importante, sino también otros aspectos primarios como la representación de la información, los modelos que abstraen los fenómenos que se estudian y algunas técnicas del área de inteligencia artificial. Tiene un gran potencial cuando se hace un procesamiento de datos a gran escala.

Como es posible observar, las aplicaciones de la minería de datos son cuantiosas y muy variadas. Por otro lado, el conocimiento que se descubre va más allá de lo que la estadística podría arrojar como resultado. Por esta razón la minería de datos se ha aplicado como herramienta, prácticamente en todas las áreas del conocimiento desde hace varios años. El resultado ha sido un impacto positivo en la capacidad de las empresas para entender los fenómenos que la rodean.

Actividad de repaso

Instrucciones: Analice los siguientes problemas y elabore las soluciones correspondientes.

Problema 1. Combinaciones

Karen está a punto de salir, pero no puede decidir qué ropa ponerse, ella tiene tres faldas y cinco blusas. Elabore un programa que determine todas las posibles combinaciones falda-blusa que tiene para escoger y que las muestre en forma de lista.

Problema 2. Piezas lego

Se tiene un conjunto de piezas de lego para construir. Sólo existen dos tipos de piezas, cubos simples y ruedas. Para formar un auto se requieren 10 cubos y 4 ruedas. Para formar un avión se requieren 15 cubos y 2 ruedas. Dado el número de cubos y ruedas, determinar cuál objeto conviene construir si sabemos que se debe construir el mayor número de objetos posible y todos ellos deben ser iguales.

Problema 3. Relación numérica

Dados tres números determinar si la suma de cualquier pareja de ellos es igual al tercer número. Si se cumple esta condición escribir: “Existe relación” y mostrar la relación. De lo contrario escribir: “No se relacionan”. Por ejemplo, 3 9 6 sí tienen relación puesto que 6+3=9.

Problema 4. Envasado

Una empresa se encarga de producir y envasar leche para consumo humano. Actualmente tienen varios contenedores con capacidad de hasta 100 litros y empaques comerciales

estampados para venta de 5 litros, 3 litros y 1 litro de capacidad. Elaborar un programa que dada la cantidad de litros que tiene un contenedor (un número mayor a 0 y menor o igual a 100). Determinar cuántos empaques de 5 litros pueden llenarse, cuántos de 3 litros y cuántos de 1 litro. Primero deben llenarse tantos empaques de 5 litros como sea posible, ya que es el producto más solicitado; luego, los de 3 litros; y, finalmente, los de 1 litro.

Problema 5. Supermercado

En un supermercado se requiere de un sistema que automatice el cálculo de la cuenta de los clientes. El sistema deberá preguntar al cajero cuántos artículos comprará el cliente, enseguida deberá solicitar la clave y el precio de cada uno de los productos que el cliente llevará. Si el cliente compra más de $10,000.00, se le hará un descuento del 10%. Finalmente, deberá mostrar la suma total, preguntar el importe del billete con el que paga el cliente y calcular el cambio que el cajero debe dar. Además del monto del cambio, debe indicar la cantidad de monedas de cada denominación que el cajero debe entregar, el algoritmo debe minimizar el número de monedas total.

Problema 6. Peces

Se ha instalado un sensor submarino para contar el número de peces que pasan bajo él en un lago. El sensor tiene un temporizador que envía el carácter “T” al procesador cada segundo. De detectarse la presencia de un pez, el sensor envía el carácter “P” al procesador. Al finalizarse el período de temporización, a la “T” le sigue inmediatamente una F. Como ejemplo, un flujo de datos normal podría ser:

T P T T T P P T P P P P T P T F

De ello se desprende que la primera detección de un pez tuvo lugar en el segundo 2. Luego, en el 6 y 7 pasaron dos peces bajo el sensor, etc. Construya un programa que introduzca los datos procedentes del sensor y que produzca la siguiente salida:

1. Número de segundos que el sondeo estuvo funcionando.
2. Total de peces que pasaron bajo el sensor.
3. Mayor número de peces en segundos consecutivos que pasaron por el sensor.

Los datos que constituirían la salida del ejemplo anterior serían:

1. El sondeo duró 16 segundos.
2. Un total de 8 peces pasaron bajo el sensor.
3. El mayor número de peces que pasaron en segundos consecutivos bajo el sensor fue de 4.

Problema 7. Ciclos

Escribir un programa que visualice el siguiente triángulo isósceles dado el tamaño de la base, el programa debe verificar que se trate de un número impar.

Problema 8. Goldbach

En 1742, el ocurrente de Christian Goldbach conjeturó que cualquier número par mayor que dos podía ser escrito como la suma de dos números primos, por ejemplo 16 = 3 + 13. Elabore un programa que le pida al usuario, a través del teclado, un número positivo N menor que 100. Enseguida, el programa deberá decidir si ese número cumple con la hipótesis de Goldbach o no. En caso de que cumpla con la conjetura deberá indicar cuáles son las sumas de números primos que dan como resultado N.

Problema 9. Vueltas al lago

Tres amigos pasean en bicicleta por un camino que bordea un lago. Para dar una vuelta completa, Pepe tarda 15 minutos; Alan, 18 minutos; y, Juan, 20 minutos. Parten juntos siempre a las 4:00 p.m. y acuerdan interrumpir el paseo la primera vez que los tres pasen, simultáneamente, por el punto de partida.

a. ¿Después de cuántos minutos termina el paseo?
b. ¿Cuántas vueltas dio cada uno?

Elaborar un programa que pida, a través del teclado, el tiempo que tarda cada uno en dar una vuelta y conteste a los incisos a y b (deberá seguir funcionando bien si cambian los tiempos).

Problema 10. Simulación

Una cadena multinacional de ventas por Internet está planeando sus promociones del próximo año. La idea consiste en registrar la fecha de cumpleaños del cliente al momento de comprar y darle un regalo especial al primer par de clientes cuyos cumpleaños coincidan.

El problema de esto es que los ejecutivos quieren saber de antemano cuantos premios se estarían entregando por día para considerarlo en el presupuesto. Para ello se requiere saber cuántos clientes deben comprar en promedio para que la fecha de cumpleaños de dos de ellos coincida. Estadísticamente se sabe que, por día, 1000 clientes compran a través de la página.

El programa deberá pedir al usuario el número **N** de simulaciones que desea hacer. Cada simulación consiste en generar una serie de cumpleaños aleatorios y contar cuántos clientes deben comprar para que dos cumpleaños coincidan. El número promedio de clientes que deben comprar para ganar un regalo será el promedio de los resultados de las N simulaciones.

Por simplicidad no use fechas (dd/mm/aaa), considere que la fecha de cumpleaños de cada cliente es un número entre 1 y 356.

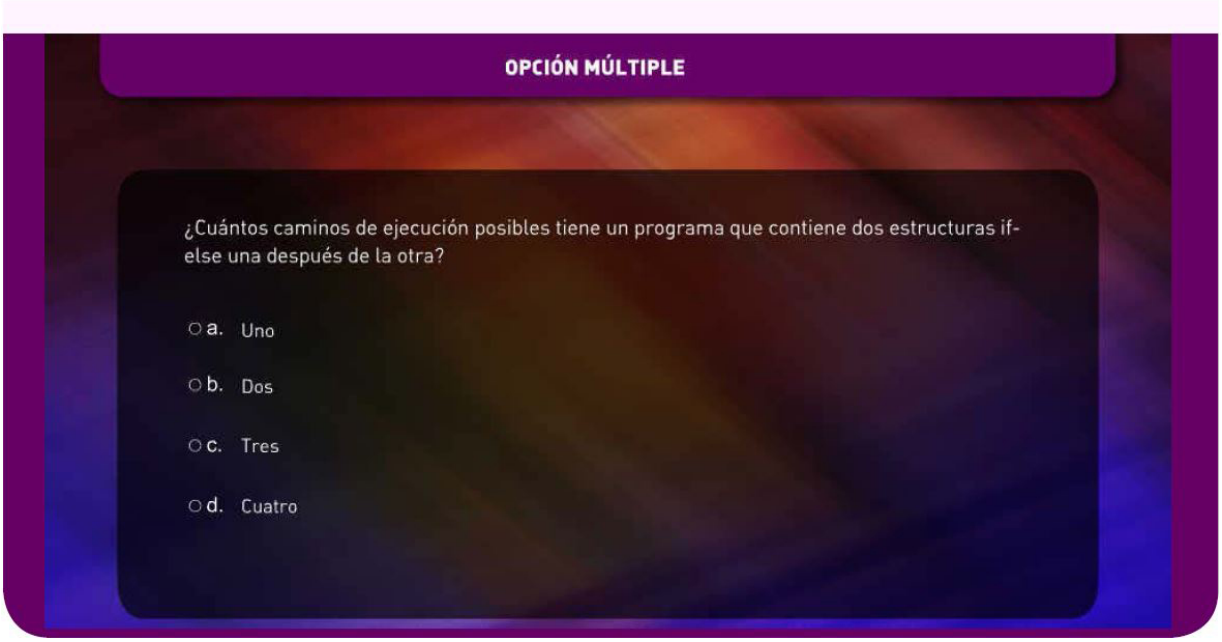
El programa deberá mostrar el número de clientes promedio que debe comprar para que un premio sea entregado y el

número de premios que se entregarían por día.

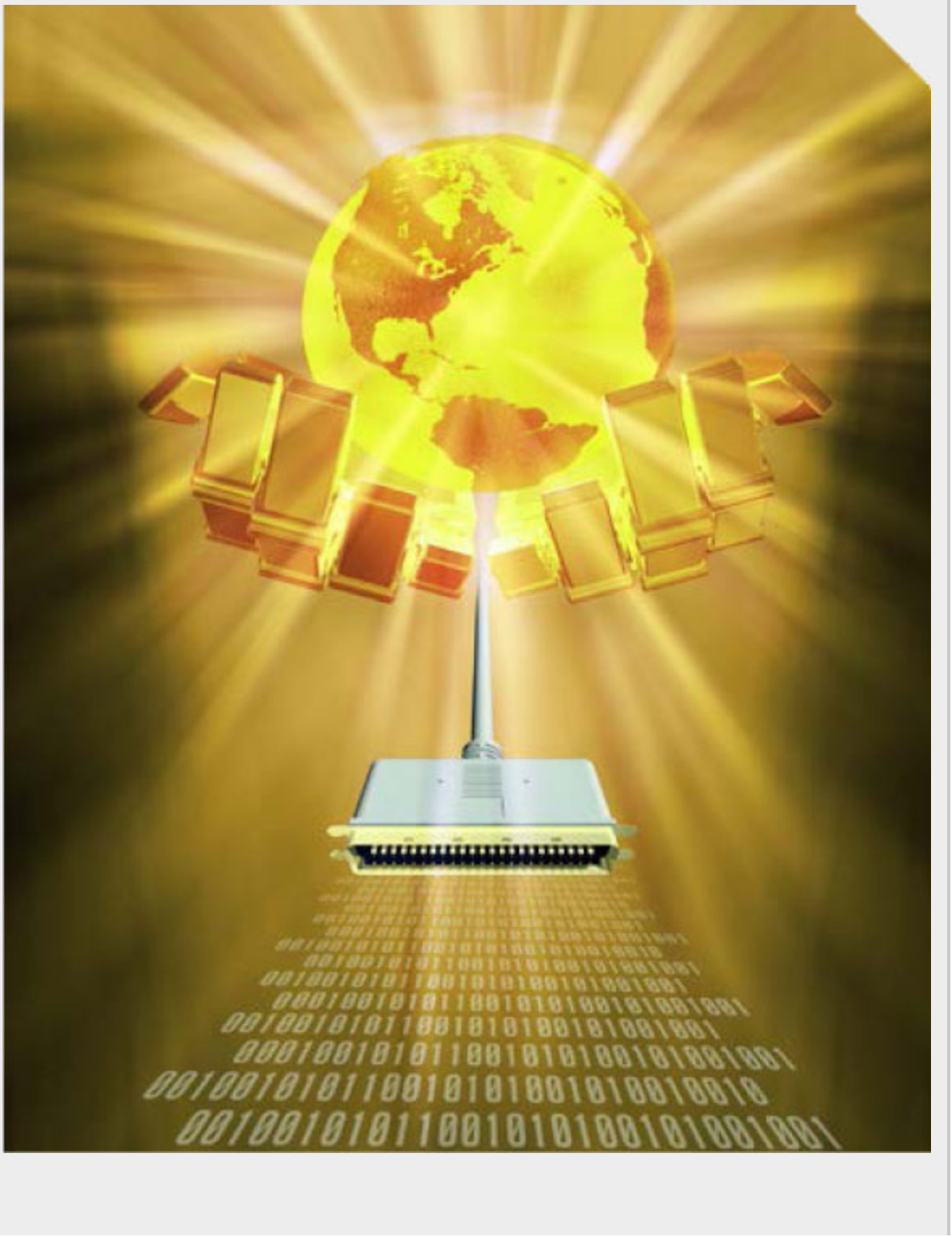
Problema 11. Menú

Elabore un programa que muestre un menú de tres opciones: “1.Mínimo común múltiplo”, “2.Máximo común divisor” y “3.Salir”. El usuario deberá poder elegir la opción deseada y el programa deberá pedirle los datos necesarios para efectuar la operación. Al elegir la opción 3 el programa deberá terminar.

Ejercicio integrador del capítulo 4



Conclusión del capítulo 4



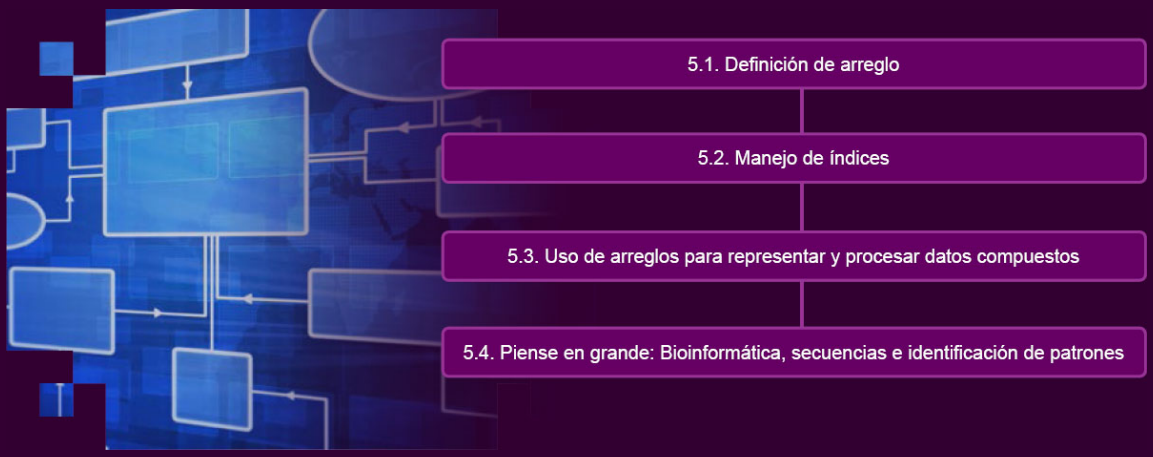
Como se ha visto a lo largo de este capítulo, las estructuras condicionales y los ciclos son dos de los elementos medulares de la programación. Sin la existencia de estas dos estructuras sería imposible concebir los sistemas computacionales que se conocen actualmente y que tanto impactan en la vida de las personas y de las empresas.

De hecho, una de las preocupaciones más grandes en la actualidad es que, dada la enorme cantidad de información que se genera, la humanidad está desperdiciando conocimiento potencialmente valioso que podría extraerse de ella. Hoy en día, aún no existen ni el poder de cómputo ni la inteligencia artificial, necesarios para analizar la información de manera autónoma y continua.

Como el resto de las áreas del conocimiento, el diseño de la tecnología debe evolucionar hacia un paradigma orientado no solamente al procesamiento de la información, sino al análisis de la misma para la generación de conocimiento.

Capítulo 5. Arreglos: almacenamiento temporal

Organizador temático



Arreglos: almacenamiento temporal

5.1 Definición de arreglo

Después del diseño de un algoritmo, un aspecto complementario, aunque no por eso menos importante, es el manejo y almacenamiento de la información que el algoritmo procesa. Dicha información puede almacenarse en estructuras de datos que residen en memoria, lo que facilita el acceso y agiliza la ejecución de los programas.

Conforme se incrementa la complejidad de los problemas por resolver, es necesario manipular una mayor cantidad de información durante la ejecución de un programa. Si bien los tipos de datos básicos proporcionan facilidades para el manejo de los datos

elementales, comienza a ser indispensable el uso de estructuras que permitan una mayor capacidad de almacenamiento.

Hasta este punto el lector ha observado que por cada variable declarada puede almacenarse un dato, es decir, una variable de tipo int almacena sólo un dato del tipo citado; lo mismo con una variable de tipo float o una de tipo char. En términos simples, un arreglo viene a extender significativamente esta capacidad limitada de almacenamiento.

Un arreglo es una estructura de datos que puede almacenar uno o más valores del mismo tipo bajo un mismo nombre.

Con base en lo anterior surgen dos preguntas esenciales en relación con esta nueva manera de concebir el almacenamiento:

1. ¿Cómo se indica al compilador cuántos elementos se desea almacenar?
2. ¿Cómo se hace referencia a cada uno de los valores que forman parte del arreglo?

Para encontrar respuesta a esta y otras preguntas revise la Figura 5.1. En donde se presenta la sintaxis correcta para la declaración de arreglos y el mecanismo que permite acceder a cada uno de sus elementos.

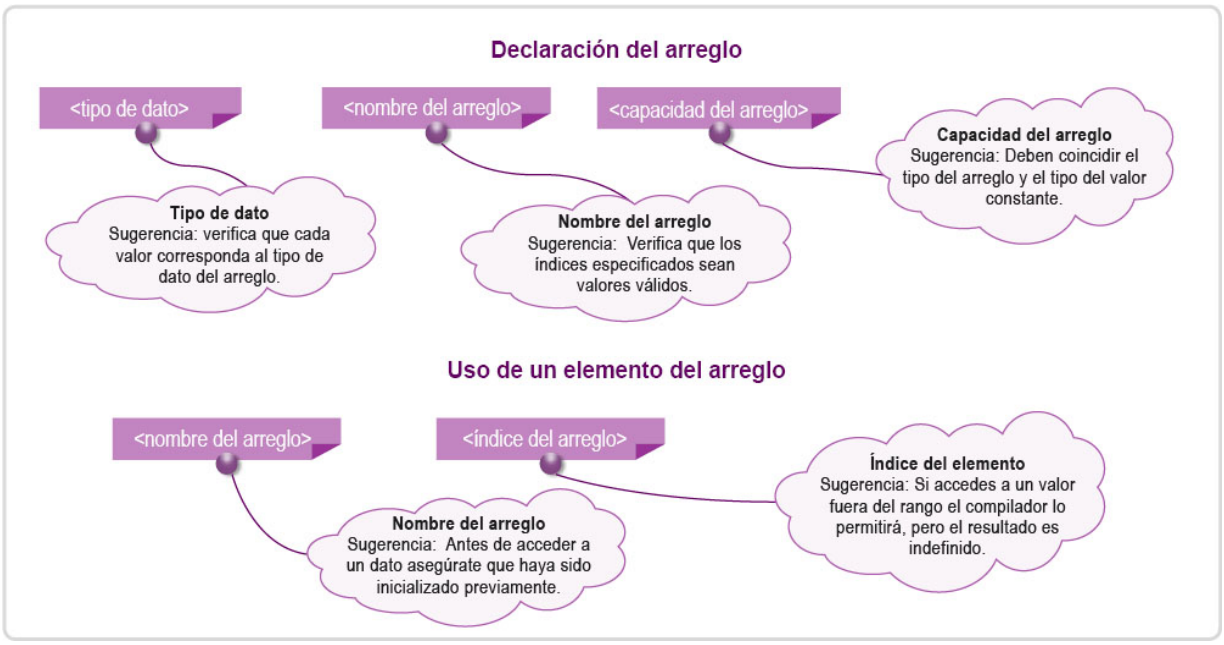


Figura 5. 1. Sintaxis y manejo de arreglos.

Como puede observarse, los arreglos son una extensión a la capacidad de almacenamiento que ofrecen las variables. Observe, en la Figura 5.2, un ejemplo de su uso.

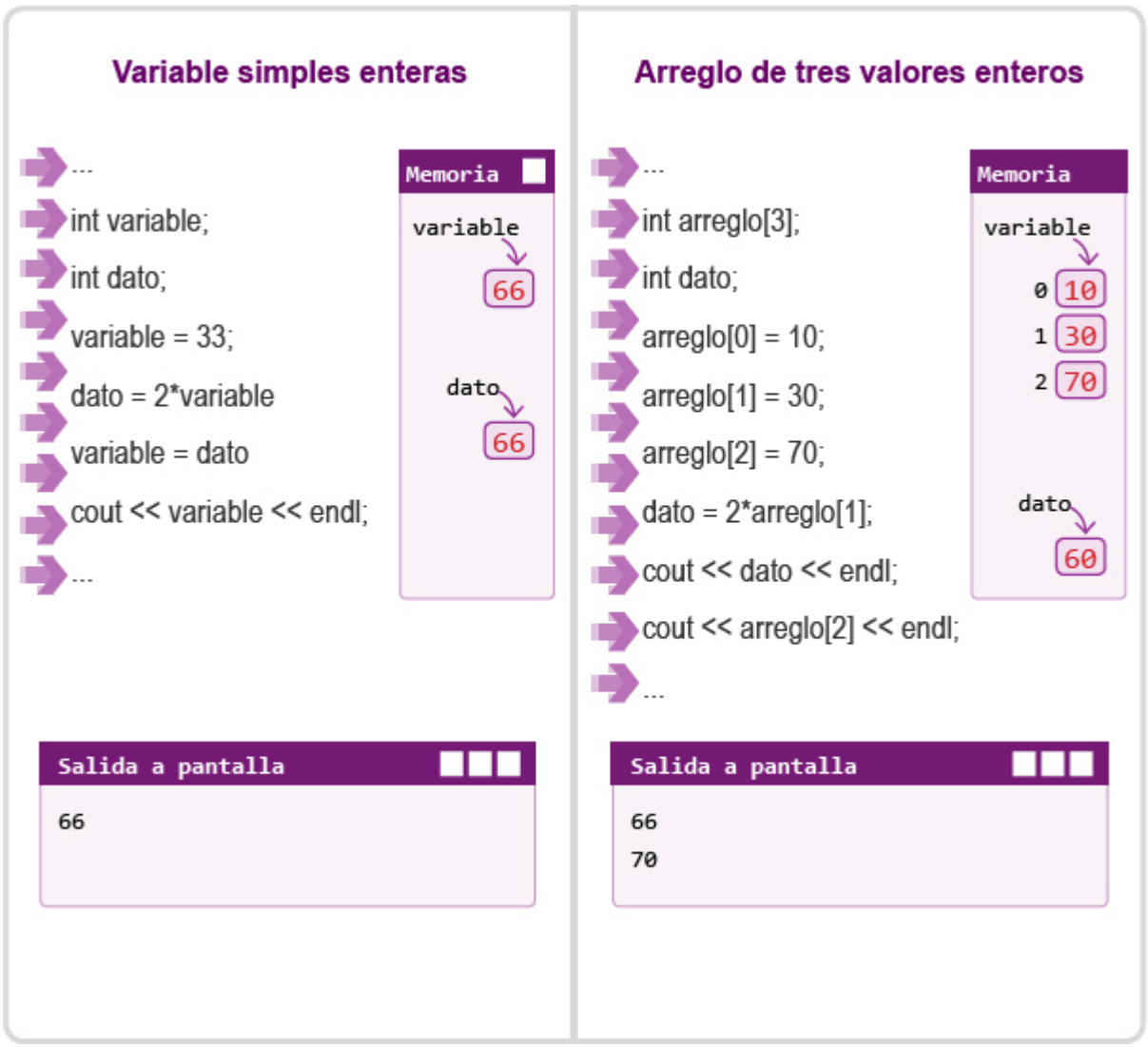


Figura 5. 2. Sintaxis y manejo de arreglos.

Es importante observar que crear un arreglo no implica que este sea inicializado. Cuando se crea un arreglo se reserva el espacio que se solicita, pero no debe asumirse que las casillas son inicializadas a algún valor en particular. De hecho, los valores almacenados son aleatorios dependiendo de lo que guardaba ese segmento de memoria antes de la creación del arreglo.

Dado lo anterior, debe quedar muy claro que una variable que representa a un arreglo se constituye entonces como una referencia a la **dirección de memoria** donde está almacenado un conjunto de datos. Ya sea que se trate de una variable simple o de un arreglo,

en ambos casos las variables no son más que referencias a las direcciones de memoria en donde se almacenan los datos correspondientes.

En el caso de los arreglos al definir el número de elementos que almacena se le está indicando al compilador el número de espacios de memoria que debe reservar para el arreglo. De esta manera los segmentos de memoria en cuestión quedan reservados para su uso exclusivo por medio de la referencia al primer elemento del arreglo.

En el caso particular de C++ existe una alternativa para la declaración de un arreglo. Esta alternativa no precisa del tamaño indicado por medio de un número entero, sino que recibe una lista de los valores iniciales para cada casilla. El tamaño se calcula, entonces, a partir del número de elementos que se indiquen entre llaves y separados por comas. En la Figura 5.3 se observa el manejo de memoria y la sintaxis de inicialización de un arreglo.

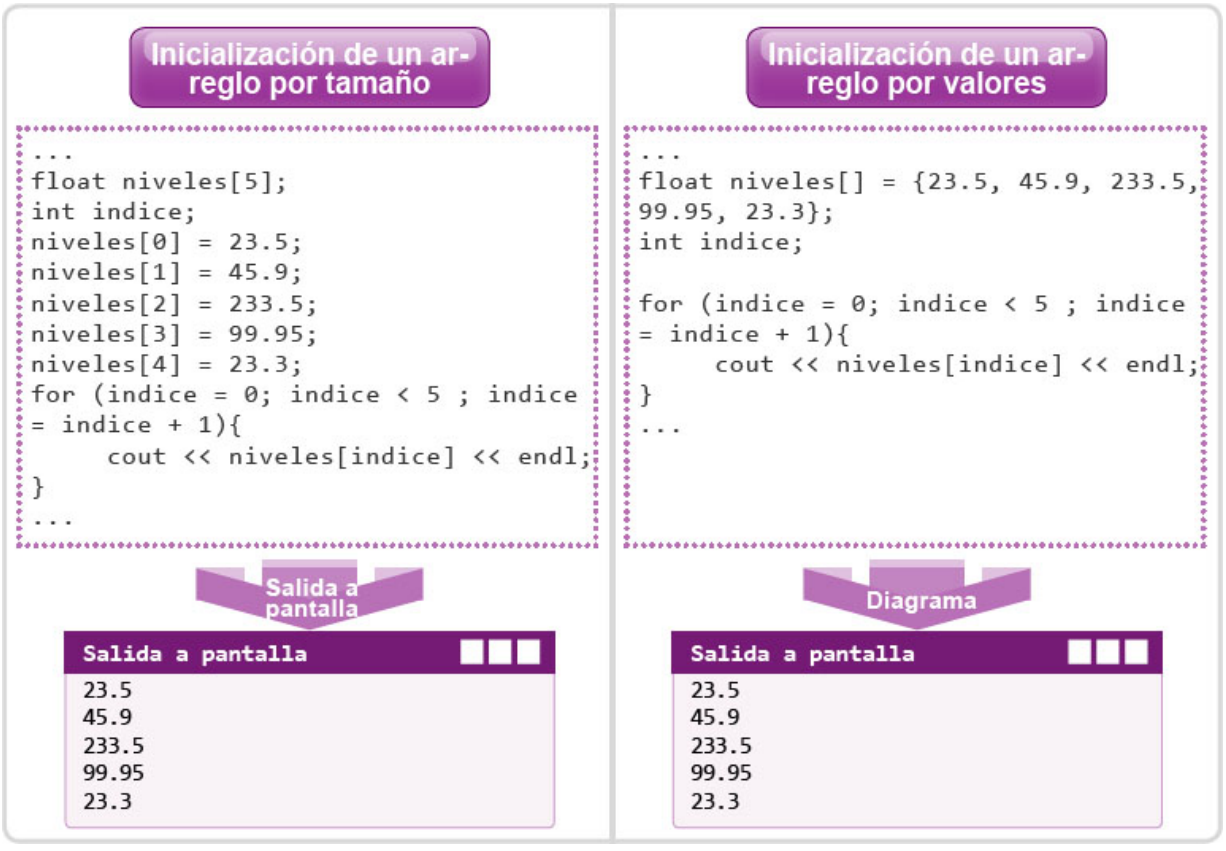


Figura 5. 3. Manejo de memoria e inicialización de arreglos.

Sin importar cuál sea la manera en que se declare un arreglo, lo significativo de este radica en las enormes posibilidades que proporciona en relación al manejo de la información. No es para nada una casualidad que el acceso a los elementos que lo integran sea por medio de referencias numéricas, ya que de esta forma puede accederse a ellos de manera automática para procesarlos.

5.2. Manejo de índices

Un índice es un valor numérico que representa la posición en la que está un dato dentro de un arreglo. El índice va de 0 a N-1, donde N es la capacidad del arreglo. En la Figura 5.4 puede identificar algunas operaciones básicas que se realizan con los elementos de un arreglo.

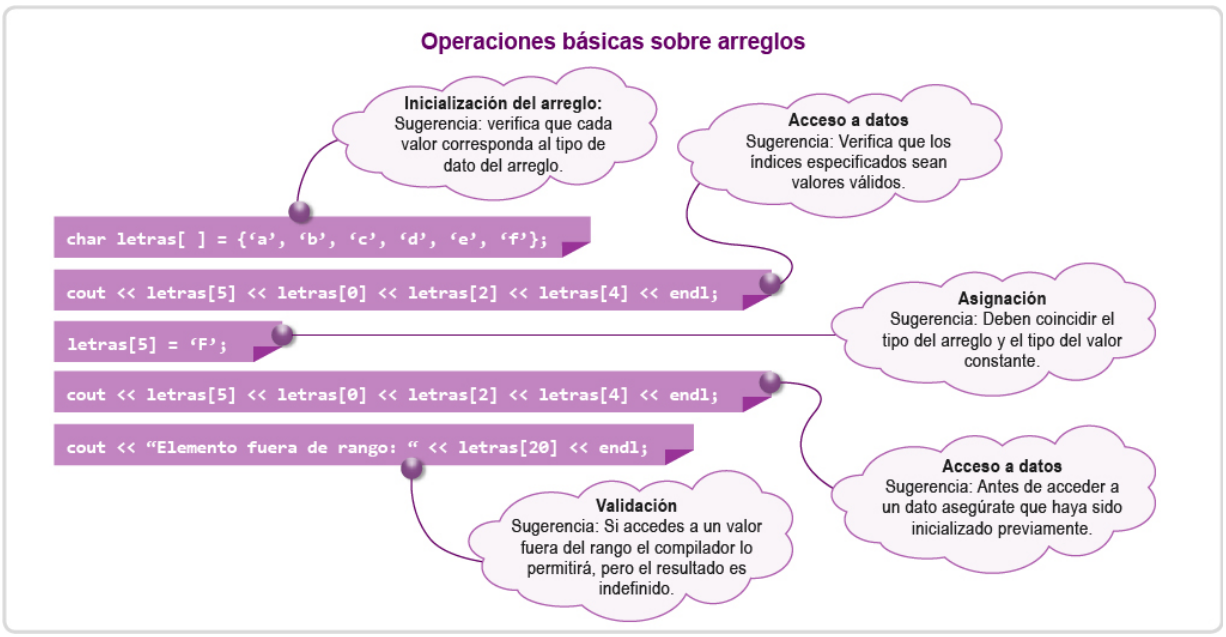


Figura 5. 4. Ejemplos del manejo de índices en arreglos.

Como puede observarse, el procesamiento automático de los elementos de un arreglo es la clave para que estos resulten útiles en la resolución de problemas. En la Figura 5.5 podrá observar el papel que juegan los arreglos en la resolución de un problema específico.

Problema

Elabore un programa que realice una simulación de 1000 tiradas de un dado. Al finalizar la simulación, muestra la frecuencia con la que se obtiene cada resultado posible.

Análisis del problema

Datos de entrada:

- Ninguno

Datos de salida:

- El número de veces que se obtuvo cada resultado posible.

Planteamiento del problema:

Realizar la simulación de 1000 tiradas de un dado y determinar la frecuencia con que se obtiene cada resultado posible.

Diseño del algoritmo

Consideraciones de diseño:

- Se necesitan variables contadores para cada resultado de los seis posibles.
- Se puede usar un arreglo de contadores de tal manera que cada casilla represente la frecuencia con la que sale cada resultado.
- Todos los elementos del arreglo deben comenzar en cero.
- El valor aleatorio puede obtenerse con un generador de números pseudoaleatorios.

Diseño del algoritmo

```
Algoritmo dado
Variables
   i, n, dado
   Dimensión frecuencias[6]
Inicio
   Para i←0  hasta 999 Con Paso 1 Hacer
      dado = random() mod 6
      frecuencias[dado] ← frecuencias[dado] + 1
   Finpara
   Para i←0  hasta 4 Con Paso 1 Hacer
      Escribir "Resultado: ",  i+1,
               " frecuencia ", frecuencias[i]
   Finpara
Fin
```

Elaboración del programa

```
#include<iostream>
#include<ctime>
using namespace std;

int main(){
   int frecuencias[6];
   int i, n, dado;

   srand(time(0));
   for (i=0 ; i<6 ; i++) frecuencias[i] = 0;
   for (n=0 ; n<1000 ; n++){
      dado = rand() % 6;
      frecuencias[dado] = frecuencias[dado] + 1;
   }
   for (i=0 ; i<6 ; i++)
      cout << "Resultado: " << i+1 << ", frecuencia "
           << frecuencias[i] << endl;
   return 0;
}
```

Verificación del programa

Salida a pantalla

```
Resultado: 1, frecuencia 164
Resultado: 2, frecuencia 164
Resultado: 3, frecuencia 171
Resultado: 4, frecuencia 170
Resultado: 5, frecuencia 163
Resultado: 6, frecuencia 168
```

Figura 5. 5. Solución al problema de la frecuencia del resultado con un dado.

Ya que el problema que se presenta en la Figura 5.5 trata sobre un generador de números aleatorios, es de esperarse que conforme crece el número de tiradas, la cantidad de veces que sale cada resultado se uniformice. Ahora bien: ¿qué sucedería si se agrega un dado más a la ecuación? Al revisar la Figura 5.6 podrá averiguarlo.

Problema

Elabore un programa que realice una simulación de 1000 tiradas de un par de dados. Lo que se busca saber es cuáles son los resultados que se obtienen con mayor frecuencia.

Análisis del problema

Datos de entrada:

- Ninguno

Datos de salida:

- El número de veces que se obtuvo cada resultado posible.

Planteamiento del problema:

Realizar la simulación de 1000 tiradas de dos dados y determinar la frecuencia con que se obtiene cada resultado posible.

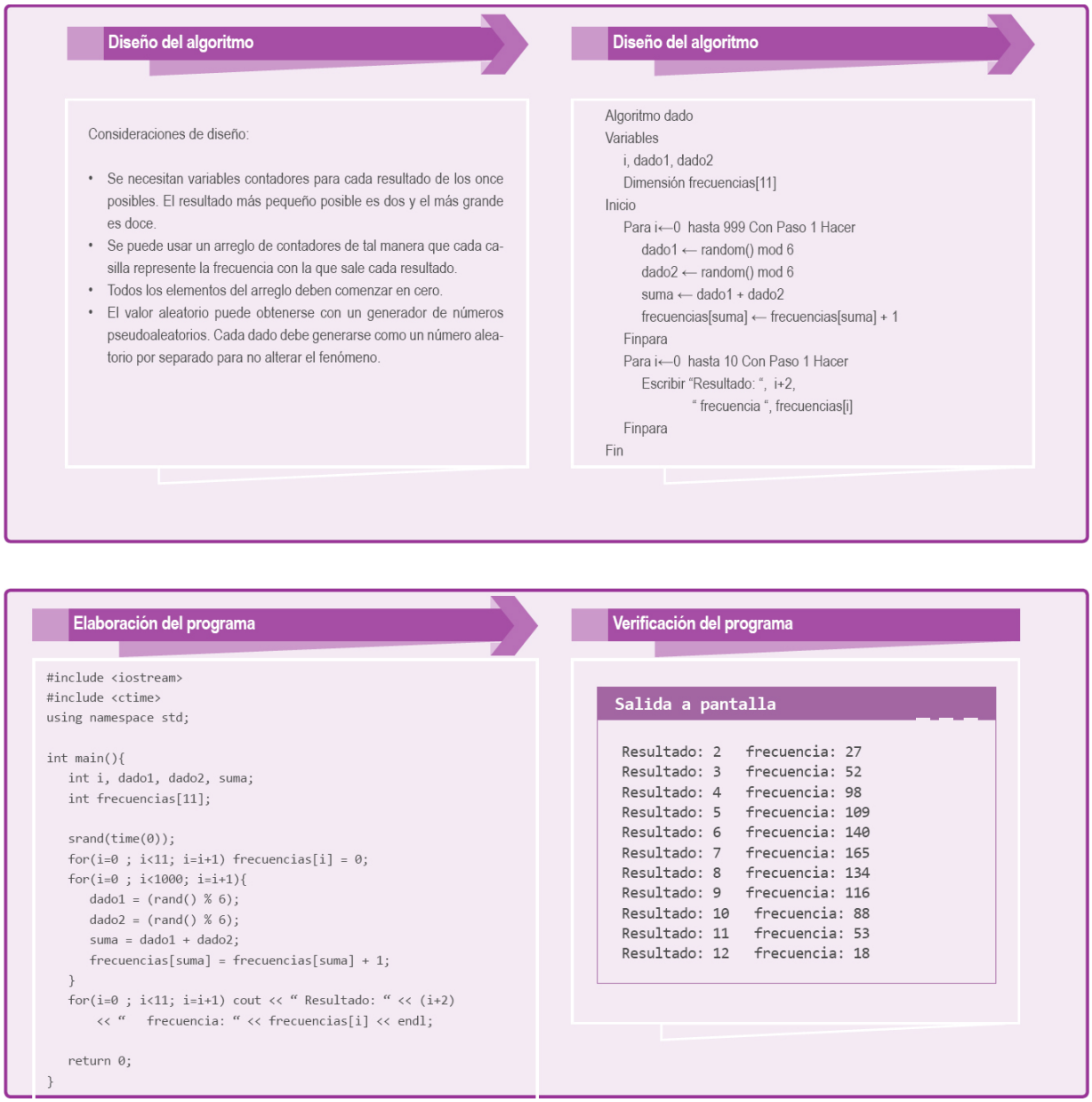


Figura 5. 6. Solución al problema de la frecuencia de resultados con dos dados.

Al observar el problema que se presenta en la Figura 5.6 salta a la vista que el resultado de esta simulación no es uniforme. Si se analizan las combinaciones que pueden dar origen a cada uno de los posibles resultados, se ve claramente que no todos tienen el mismo número de combinaciones posible. De ahí la discrepancia entre la frecuencia del resultado siete contra la frecuencia de los resultados doce o dos.

La facilidad para declarar arreglos de gran capacidad no quiere decir que el uso de los arreglos sea indiscriminado. Como en casi todas las estructuras, instrucciones y elementos de programación, de su uso racional se desprenden ventajas o desventajas. En la Figura 5.7 podrá identificar algunas consideraciones importantes para la solución de un problema mediante arreglos.

Problema

Elabore un programa que simule el funcionamiento de una tómbola, de tal forma que genere 6 números aleatorios distintos dentro del rango de 0 a 99.

Análisis del problema

Datos de entrada:
- Ninguno

Datos de salida:
- Los seis números resultantes del sorteo.

Planteamiento del problema:
Simular el funcionamiento de una tómbola de la cual se extraen 6 números diferentes en un rango de 0 a 99.

Diseño del algoritmo

Consideraciones de diseño:

- Cada nuevo número puede obtenerse con un generador de números aleatorios.
- Para determinar si dicho número ya salió es necesario conocer los números que han salido previamente.
- Lo anterior puede hacerse por lo menos de dos maneras:
  - Definir un arreglo de 100 elementos e ir marcando aquellos que ya salieron de manera binaria ("ya salió", "no ha salido")
  - Definir un arreglo de 5 elementos e ir almacenando los números que han salido. Al generar cada nuevo número buscar en la lista para verificar si ya existe.

Diseño del algoritmo

```
Algoritmo tómbola
Variables
    turno, numero
    Dimensión marcas[100]
Inicio
    Para turno← 0  hasta 99 Con Paso 1 Hacer
        marcas[turno] ← false
    Finpara
    Para turno←0  hasta 5 Con Paso 1 Hacer
        numero ← generarNumero(marcas)
        Escribir "Número elegido: ", numero
    Finpara
Fin
```

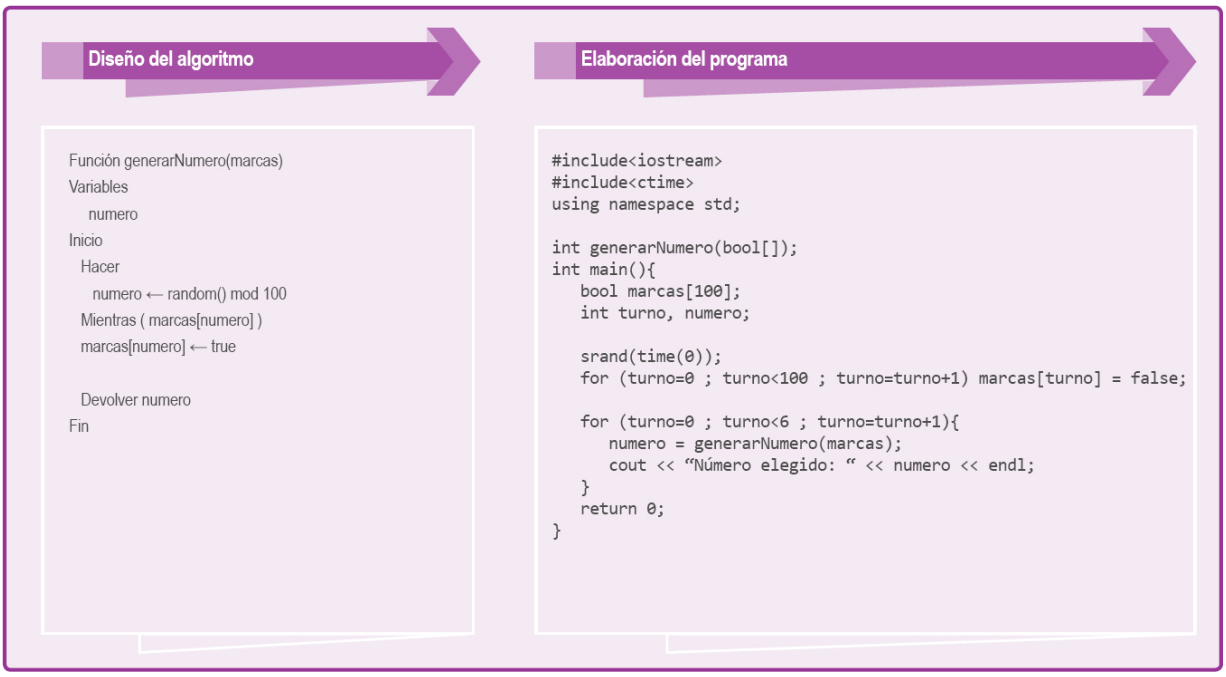


Figura 5. 7. Solución al problema de la tómbola.

La solución que se plantea en el problema de la Figura 5.7 tiene la característica de ser muy eficiente en tiempo. Es decir, realiza el número de operaciones mínimo para verificar si un nuevo número generado aleatoriamente ha salido antes o no. Sin embargo, no es la más eficiente en términos del almacenamiento que se emplea.

Revise ahora una solución que busque reducir el espacio de almacenamiento requerido para lograr el mismo resultado; revise la Figura 5.8 para tal efecto.

Problema

Elabore un programa que simule el funcionamiento de una tómbola y que genere 6 números aleatorios distintos dentro del rango de 0 a 99.

Análisis del problema

Datos de entrada:
- Ninguno

Datos de salida:
- Los seis números resultantes del sorteo.

Planteamiento del problema:
Simular el funcionamiento de una tómbola de la cual se extraen 6 números diferentes en un rango de 0 a 99.

Diseño del algoritmo

Consideraciones de diseño:

- Cada nuevo número puede obtenerse con un generador de números aleatorios.
- Para determinar si dicho número ya salió es necesario conocer los números que han salido previamente.
- Lo anterior puede hacerse por lo menos de dos maneras:
  - Definir un arreglo de 100 elementos e ir marcando aquellos que ya salieron de manera binaria ("ya salió", "no ha salido")
  - **Definir un arreglo de 5 elementos e ir almacenando los números que han salido. Al generar cada nuevo número buscar en la lista para verificar si ya existe.**

Diseño del algoritmo

```
Algoritmo tómbola
Variables
    Dimension sorteo[6]
    k, numero
Inicio
    Para k←0  hasta 5 Con Paso 1 Hacer
        numero ← generarNumero(sorteo)
        Escribir "Número elegido: ", numero
    Finpara
Fin
```

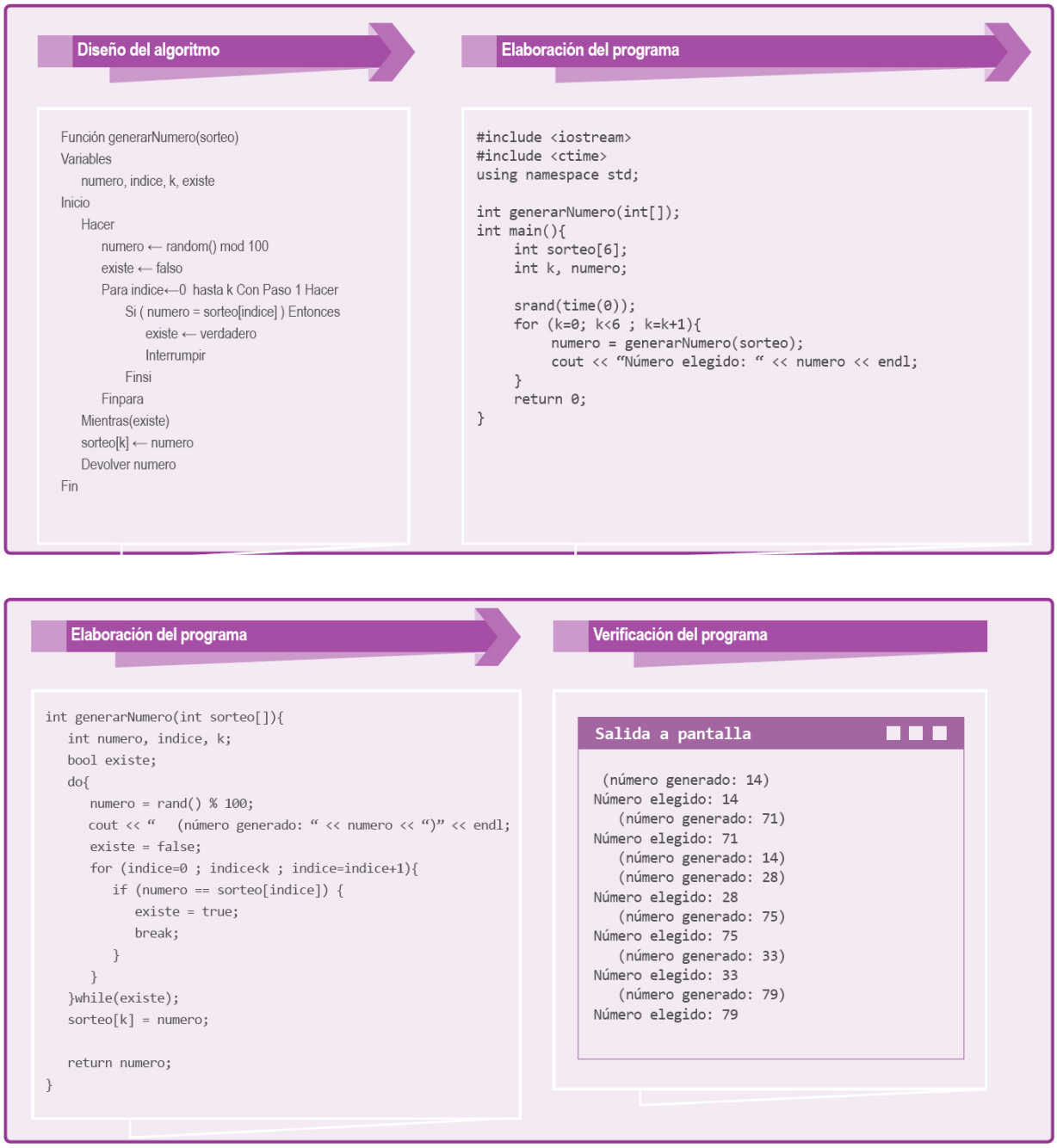


Figura 5. 8. Solución alterna al problema de la tómbola.

La reflexión más importante que el lector debe hacer en relación a los problemas anteriores está directamente relacionada con la siguiente pregunta: ¿cuál de las dos soluciones es la más eficiente? Si no tiene una respuesta contundente de inmediato no se preocupe; este dilema ha atormentado a los especialistas del área de las ciencias computacionales durante años.

Sucede que la pregunta “¿cuál de las dos soluciones es la más eficiente?” está incompleta ya que la eficiencia puede medirse de dos maneras: **eficiencia temporal o eficiencia espacial** (de almacenamiento). Si lo primordial es la eficiencia temporal, entonces será mejor la solución que realiza menos operaciones. Por otro lado, lo importante es la eficiencia de almacenamiento será mejor la solución que utilice un menor espacio. En la Figura 5.9 se identifican las cuatro principales aristas de la eficiencia en términos de soluciones computacionales.

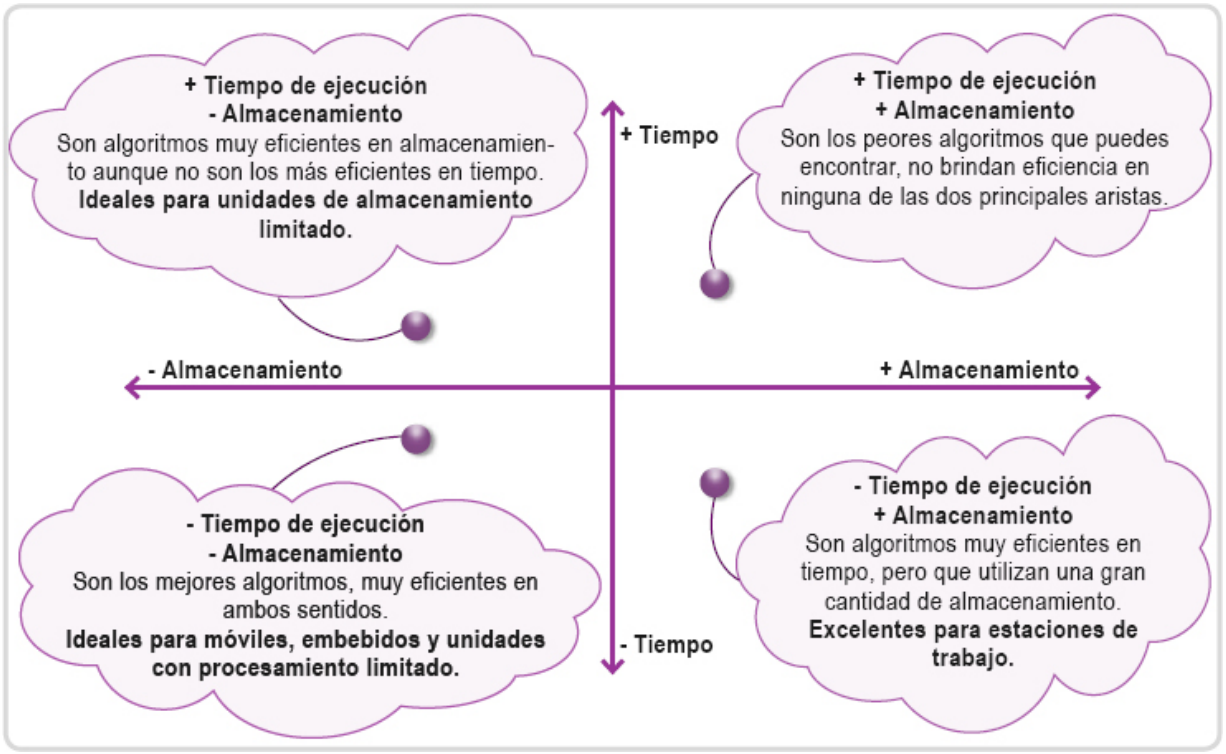


Figura 5. 9. Consideraciones de eficiencia en el diseño de algoritmos.

Un profesionista que inicia en el terreno de la programación debe considerar las restricciones asociadas con la problemática por resolver. No es lo mismo diseñar un algoritmo que será ejecutado en una PC a diseñar un algoritmo que se ejecutará en la **computadora de abordo** de un automóvil. La capacidad de procesamiento y almacenamiento varían de una opción a otra por lo que la solución más eficiente no está en función de la solución en sí misma; la solución más eficiente aprovecha al máximo los recursos que le presenta el medio en función del tamaño del problema que resuelve.

5.3. Uso de arreglos para representar y procesar datos compuestos

Hasta este punto sólo se ha trabajado con arreglos unidimensionales, es decir, con arreglos que almacenan datos de manera lineal. Sin embargo, el mismo mecanismo diseñado para los arreglos unidimensionales es extensible para su uso en dos, tres o más dimensiones. Basta con agregar otro componente a la declaración del arreglo para tener una dimensión adicional. En el Anexo 4, ubicado en la barra lateral, se aborda el tema de **arreglos** de más de una dimensión.

5.4 Piense en grande: bioinformática, secuencias e identificación de patrones

Como se ha visto en este capítulo, los arreglos son estructuras de datos muy versátiles, capaces de almacenar grandes volúmenes de información. Aunque los ejemplos ilustrativos del capítulo presentan al lector las facilidades en cuanto al manejo de los datos, no hay nada como los ejemplos del mundo real para ilustrar este punto.

En los últimos años, han surgido áreas como la *bioinformática* que se han visto beneficiadas enormemente por el desarrollo de las ciencias computacionales. Algoritmos que fueron diseñados para el reconocimiento de patrones hoy se utilizan en el reconocimiento de cadenas o subestructuras genéticas. En el video que se presenta en la barra lateral puede observarse un ejemplo de lo anterior.

En el mundo actual es frecuente el trabajo multidisciplinario y las tecnologías computacionales son una de las disciplinas que más relación tienen con otras. La biología, la animación, el análisis predictivo, la medicina, el ambiente deportivo y muchas otras áreas se han visto favorecidas por el desarrollo de algoritmos, soluciones y estructuras que han permitido avances en cada una de estas áreas.

Actividad de repaso

Problema 1. Grupo de alumnos

Hay 10 alumnos en un grupo. Determine el promedio de sus calificaciones en la materia de programación y el porcentaje de aprobados y reprobados.

Problema 2. Licitaciones

Se tiene una licitación para la construcción de un hospital. Hay seis empresas concursando y se desea determinar cuál cotiza los precios más altos para la obra y cuál los precios más bajos.

Problema 3. Invertir

Elabore un programa que dada una palabra, la escriba al revés (bienvenido → odinevneib).

Problema 4. Lotería

Imprima una hoja de lotería de seis columnas por cuatro filas y que en cada casilla tenga los números del 1 al 24.

Problema 5. Correspondencia

Dada una matriz de 3x3 y dos fichas colocadas en dos casillas aleatorias (0 para indicar que no hay ficha y 1 para indicar que sí hay ficha), elabore un programa que determine si están en la misma línea o en la misma columna.

Problema 6. Invertir matriz

Imprima una matriz de n filas y m columnas en forma invertida.

Entrada:	Resultado:
[1 2 3]	[3 2 1]
[4 5 6]	[9 8 7]

[7 8 9] [6 5 4]
[1 2 3] [3 2 1]

Problema 7. Diagonal

Imprima la diagonal principal de una matriz cuadrada, es decir de *n* x *n* columnas.

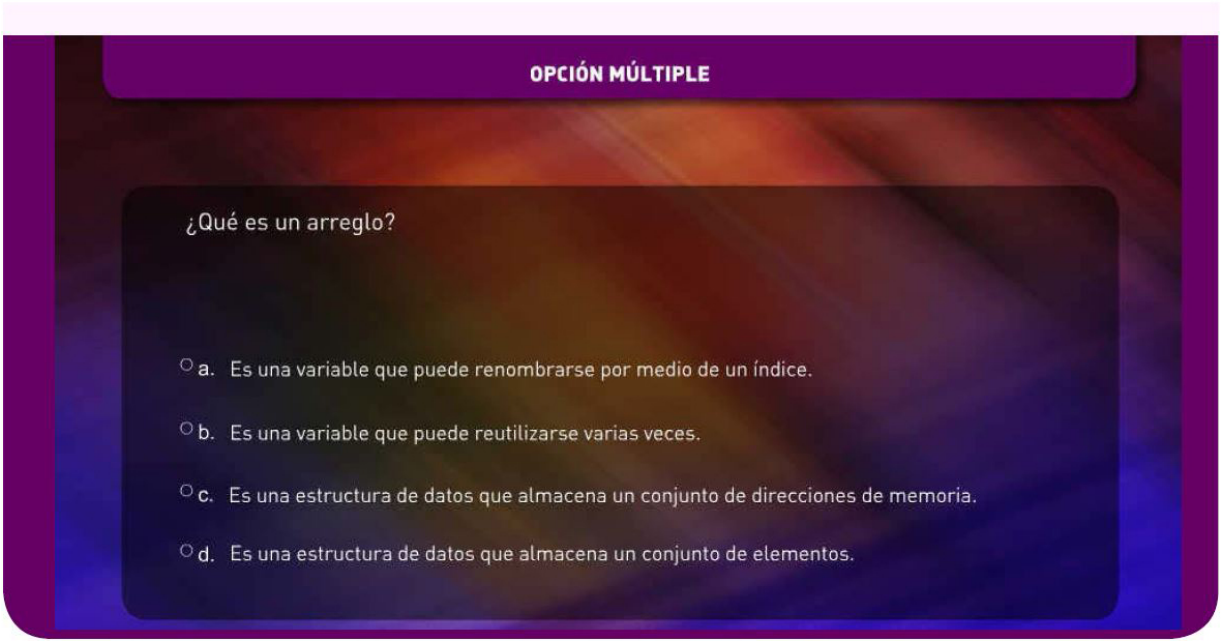
Entrada:	Resultado:
[1 2 3] [4 5 6] [7 8 9]	[1 5 9]

Problema 8. Arreglos

Dados dos arreglos de diferente tamaño cuyos elementos se encuentran en orden ascendente, elabore un programa que combine los arreglos en un sólo arreglo que mantenga los elementos.

Entradas	Salida:
int a1[6] = {1, 4, 6, 8, 10, 80}; int a2[5] = {2, 3, 7, 90, 100};	1, 2, 3, 4, 6, 7, 8, 10, 80, 90, 100

Ejercicio integrador del capítulo 5



Conclusión del capítulo 5

En la actualidad es habitual hablar de eficiencia. Aunque bien es cierto que el término tiene una connotación temporal en este capítulo se ha observado que la eficiencia puede tener múltiples aristas. En el caso de la programación dos de ellas son trascendentales: la eficiencia temporal y la eficiencia espacial.

El alcance de un objetivo desde estos dos tipos de eficiencia, aunado a la gama de posibilidades que ofrecen las herramientas modernas de desarrollo, han convertido a las tecnologías computacionales en una herramienta muy útil en la experimentación y el desarrollo de otras disciplinas.

Cuando se dan los primeros pasos en el mundo de la programación, es difícil visualizar el enorme potencial de comprensión y creación que se presenta justo ante los ojos. Pero una vez que se toma conciencia de la poderosa herramienta que se tiene en las manos, resulta difícil resistirse a saber un poco más sobre las diferentes implicaciones que esta actividad racional y analítica ofrece.

Capítulo 6. Cadenas de caracteres

6.1 Definición de cadena de caracteres

Desde hace un par de décadas los procesadores de texto han sido una de las aplicaciones computacionales más utilizadas, millones de personas alrededor del mundo no conciben la computadora sin esta aplicación muchas otras tuvieron su primer contacto con una computadora justamente a través de ésta.

Las empresas almacenan y procesan información de diferente naturaleza; si bien un gran porcentaje de la información que se procesa es numérica, hay un universo de aplicaciones que almacenan y trabajan datos como nombres, direcciones, descripciones y hasta textos literarios completos.

A fin de satisfacer la necesidad de almacenamiento de información, cada lenguaje de programación proporciona diferentes facilidades para el manejo de cadenas de caracteres o strings, por el término en inglés. En C++ existen dos formas de manipular cadenas de caracteres: la primera por medio de las funciones definidas en la biblioteca cstring, en cuyo caso las cadenas de caracteres se representan como arreglos de elementos tipo char, es decir, arreglos de caracteres. La segunda forma hace uso de la biblioteca string que está orientada al manejo de cadenas bajo el paradigma de **programación orientada a objetos**.

Para los fines de este eBook se utilizará la segunda representación basada en la biblioteca string. Ya que ofrece funciones miembro y mecanismos orientados a facilitar el manejo de este tipo de datos.

Las funciones miembro son aquellas funciones que están asociadas exclusivamente con un tipo específico de datos y reciben de manera implícita una referencia al elemento que hace el llamado a la función.

El uso de la biblioteca string es muy recomendable debido a la validación incluida en sus funciones miembro, que facilita la escritura de programas menos propensos a errores.

6.2 Operaciones sobre cadenas de caracteres

La primera operación importante consiste en conocer el número de caracteres incluidos en una cadena de texto. Ya sea para procesar sus elementos o simplemente para validar su contenido. Sin embargo, hay otras funciones miembro. En la Tabla 6.1 se muestran las principales funciones miembro del tipo string. Revise cuidadosamente la descripción de cada una y sus posibles implicaciones.

Tabla 6.1. Principales funciones para el manejo de cadenas.

Biblioteca	Función	Descripción
<string>	char at(int pos);	Devuelve el caracter en la posición *pos.*
	char* c_str( )	Devuelve una referencia a la representación de la cadena en forma de arreglo de caracteres.
	bool empty( )	Devuelve verdadero o falso dependiendo de si la cadena está vacía o no.
	int length( )	Devuelve la longitud de la cadena de caracteres.
	void clear( );	Borra el contenido de la cadena de caracteres.
	string insert(int index, string str);	Inserta la cadena str a partir de la posición *index.*
	string erase(int index, int count);	Borra count elementos de la cadena a partir de la posición *index.*
	string replace(int pos, int count, string str);	Reemplaza *count* caracteres a partir de la posición pos por la cadena *str.*
	string substr(int pos, int count);	Obtiene la subcadena compuesta por *count* caracteres a partir de la posición pos.
	int find(string str, int pos);	Obtiene la posición de la cadena *str* buscando a partir de la posición *pos.*

Como puede observarse una de las firmas de las funciones de manejo de strings hace referencia al tipo char*; el caracter asterisco indica que se trata de una referencia o **apuntador** a un arreglo de

elementos de tipo char. Esto es un aspecto heredado del lenguaje C, ya que en este las cadenas de caracteres eran simples arreglos. El concepto no es nuevo pues como se explicó en el capítulo anterior, en el caso de los arreglos, la variable que los representa contiene una referencia a la dirección de memoria en donde está almacenado el primer elemento del arreglo. En la Figura 6.1 podrá identificar las características de una cadena de caracteres y las diferentes maneras de fijar un valor inicial.

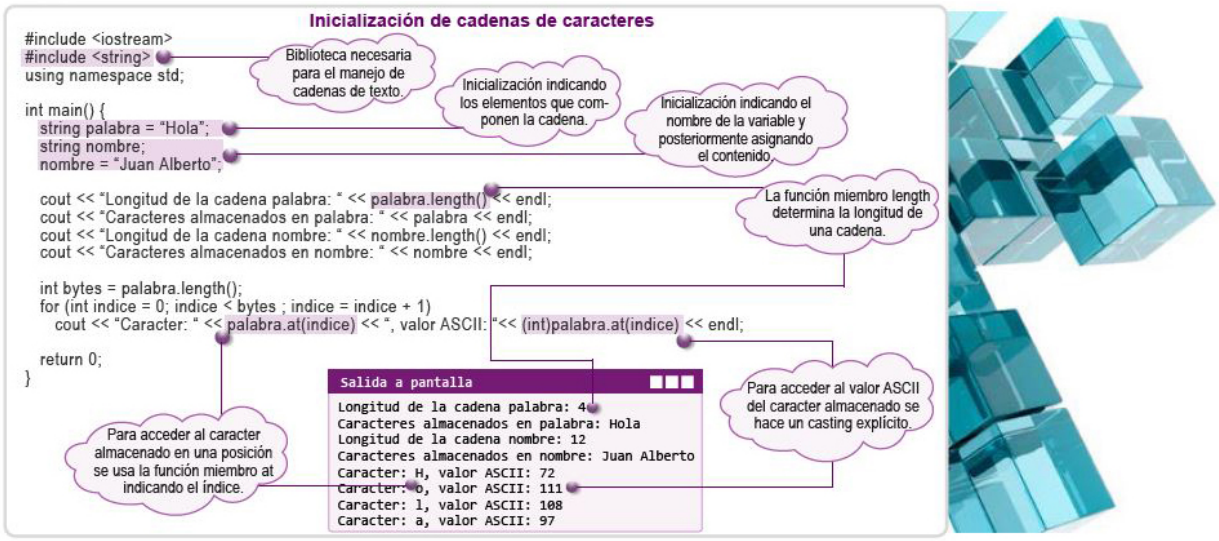


Figura 6. 1. Sintaxis e inicialización de cadenas de caracteres.

Un aspecto importante es la notación que se emplea para escribir las funciones miembro; estas funciones se colocan a la derecha del nombre de la variable, separadas por un punto. Por ejemplo, para que la función miembro *length* devuelva la longitud de una cadena llamada *nombre* deberá escribirse *nombre.length().*

Otro ejemplo es la función miembro *at,* la cual recibe como parámetro la posición del caracter que se desea obtener y devuelve el caracter mismo. En la siguiente sección se revisarán a mayor detalle las funciones miembro para la manipulación de cadenas.

Otro punto importante es que cada uno de los caracteres que componen una cadena de texto tiene una correspondencia

numérica. Esta correspondencia es estándar y se conoce como código ASCII (ver tema 2.3, Figura 2.5). Al tomar en cuenta esto último es fácil observar que pueden realizarse operaciones aritméticas con ellos pueden compararse e incluso transformarse al usar dicha representación.

Ahora bien, para obtener una cadena de caracteres por teclado puede utilizarse el flujo de entrada cin de la misma forma en que se ha hecho hasta el momento. No obstante, este tipo de entrada con formato lee solamente los caracteres introducidos y deja el caracter que representa el **salto de línea** en el flujo, lo cual podría ocasionar que una posterior entrada de datos se alimente automáticamente de este caracter; esto da la impresión de que el programa está omitiendo una entrada. Otro inconveniente de cin es que interrumpe la entrada de datos al detectar un espacio en blanco lo que podría resultar impráctico para todos los programas que utilizan cadenas de caracteres.

Una buena práctica de programación en C++ consiste en no mezclar las instrucciones get y getline, es decir, no mezclar entrada con formato y entrada sin formato.

Si se desea que el usuario sea capaz de capturar una línea completa es necesario utilizar la función *getline* la cual incluye en la cadena de caracteres los espacios en blanco que el usuario introduce. Además, procesa el caracter salto de línea sin almacenarlo y sin dejarlo en el flujo. Para observar un ejemplo del procesamiento de cadenas de caracteres como datos de entrada en un programa, revise la Figura 6.2.

Estructura condicional

```
#include  <iostream>
#include  <string>
#include  <sstream>
using namespace std;

int main()  {
        string nombre,  textoEdad,  telefono;
        stringstream  convertidor;
        int edad;

        cout  <<  "Datos de registro"  <<  endl;
        cout  <<  " Nombre...: "; getline(cin, nombre);
        cout  <<  " Edad.....: "; getline(cin, textoEdad);
        convertidor.str(textoEdad);
        convertidor  >>  edad;
        cout  <<  " Teléfono.:  "; getline(cin, telefono);

        cout  <<  "Datos capturados:"  <<  endl;
        cout  <<  "[" << nombre << "]"   <<  endl;
        cout  <<  "[" << edad << "]"   <<  endl;
        cout  <<  "[" << telefono << "]"   <<  endl;

        return 0;
}
```

Salida a pantalla

```
Datos de registro
Nombre...: Juan Valdez
Edad.....: 70
Teléfono.: 5555123456
Datos capturados:
[Juan Valdez]
[70]
[5555123456]
```

Figura 6. 2. Ejemplo del manejo de cadenas de caracteres.

Como se observa en el ejemplo, todos los valores introducidos, aun el valor *textoEdad* se obtienen mediante la función getline.

Posteriormente, se usa una variable de tipo *stringstream* para convertir el valor de *string* a *int.* Esta manera de obtener valores de entrada no sólo es más segura que otras funciones como *atoi* y *strtol,* sino que también ofrece un mecanismo para verificar errores. Si el usuario no captura un valor entero, la bandera de *estado de falla* se activa, esta bandera puede verificarse mediante la función miembro *fail* del flujo. A continuación se muestra un segmento de código usado para bloquear el flujo del programa mientras no se introduzca un valor entero válido:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main() {
string textoEdad;
int edad;
stringstream convertidor;
do{
   cout << " Edad: ";
   getline(cin, textoEdad);
   convertidor.clear(); // para reestablecer estado
   convertidor.str(textoEdad);
   convertidor >> edad;
}while(convertidor.fail());
}
```

Por supuesto, para usarlo repetidas veces dentro de un programa el segmento de código que se mostró arriba tendría que usarse dentro de una función, la cual, una vez definida, pueda usarse cuantas veces sea necesario. Dicha función quedaría de la siguiente manera:

```
int ObtenerEntero(){
string textoEdad;
int edad;
```

```
stringstream convertidor;
do{
    cout << " Edad: ";
    getline(cin, textoEdad);
    convertidor.clear();
    convertidor.str(textoEdad);
    convertidor >> edad;
    if (convertidor.fail())
        cout << "Error, introduzca un valor entero válido" <<
endl;
}while(convertidor.fail());
return edad;
}
```

Estos mecanismos de entrada de datos resultan de mucha utilidad a la hora de escribir programas confiables, ya que los valores de entrada con que estos se alimentan pueden validarse de manera

correcta. De este modo se previenen fallas en el funcionamiento de los programas, ya sea por una imprecisión al proporcionar la información de entrada o por la inserción de datos de forma malintencionada para propiciar el mal funcionamiento.

Revise ahora un ejemplo de la puesta en práctica de algunas de estas funciones. En la Figura 6.3 podrá identificar el razonamiento, consideraciones de diseño e implementación de un problema relacionado con el manejo de cadenas de caracteres.

Problema

Elabore un programa que reciba del usuario una frase y determine si la frase es un palíndromo, es decir, si puede leerse igual al derecho y al revés. El programa solo debe decir "si es un palíndromo" o "no es un palíndromo". Los espacios en blanco deben ser ignorados.

Ejemplos:
- Anilina
- Arañara
- Reconocer

Frases:
- anita lava la tina
- anula la luz azul a la luna
- arde ya la yedra
- asi le ama elisa

Análisis del problema

Datos de entrada:
- La palabra o frase a verificar

Datos de salida:
- El mensaje que indique si es palíndromo o no.

Planteamiento del problema:
Dada una palabra o frase, determinar si se trata de un palíndromo, ignorando los espacios en blanco.

Diseño del algoritmo

Consideraciones de diseño:

- Se requiere comparar el primer caracter con el último, el segundo con el penúltimo y así sucesivamente.
- No es necesario que el ciclo recorra toda la cadena, puede llegarse hasta la longitud/2
- Para acceder al caracter de la mitad derecha se puede calcular su índice a partir de longitud – 1.
- Los espacios representan un problema en el caso de las frases.
- Para evitar hacer validaciones al interior del ciclo que se usará para recorrer los elementos del arreglo, se pueden remover todos los espacios en blanco antes de hacer las comparaciones
- Siendo así, se pueden identificar dos funciones claramente: removerEspacios y esPalindromo.

Diseño del algoritmo

```
Algoritmo palíndromos
Variables
        frase
Inicio
        Escribir "Introduzca la palabra o frase: "
        Leer frase

        frase = removerEspacios(frase)
        Si (esPalindromo(frase)) Entonces
                Escribir "Sí es palíndromo"
        Sino
                Escribir "No es palíndromo"
        Finsi
Fin
```

Diseño del algoritmo

```
Función removerEspacios ( cadena )
Variables
        indice
Inicio
        Mientras ( (indice ← cadena.buscar(" ")) >= 0) Hacer
                cadena ← cadena.borrar(indice, 1)
        Finmientras
        Devolver cadena
Fin
```

Diseño del algoritmo

```
Función esPalindromo ( cadena )
Variables
        indice, longitud, izquierdo, derecho
Inicio
        longitud ← cadena.longitud()
        Para indice ← 0 hasta longitud/2 con Paso 1 Hacer
                izquierdo ← cadena.en(indice)
                derecho ← cadena.en(longitud – indice – 1)
                Si (izquierdo != derecho) Devolver falso
        Finpara
        Devolver verdadero
Fin
```

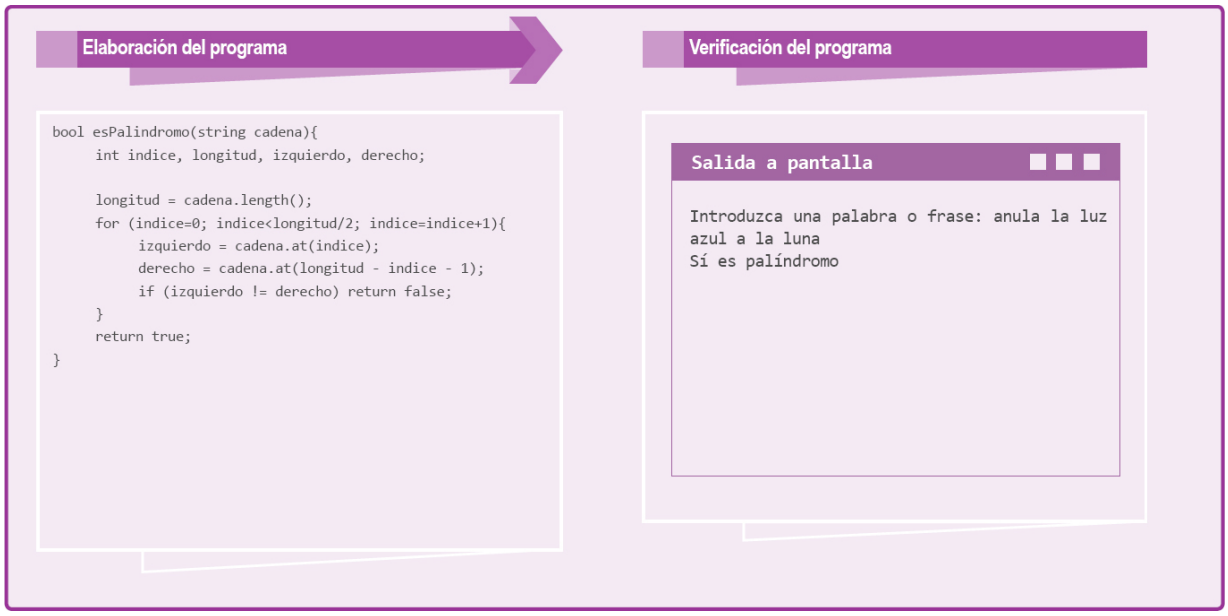


Figura 6. 3. Solución al problema de las palabras y frases palíndromos.

Los dos errores más comunes cuando se trabaja con cadenas de caracteres son la violación del rango válido de una cadena y no asignar a la misma variable string el resultado de las operaciones *find, erase, replace* y demás funciones miembro.

- En el primer caso, el problema se da principalmente por acceder a los caracteres de la cadena usando la notación arreglo[índice] y no la función miembro at. La diferencia es sustantiva, ya que mientras la notación arreglo[índice] no valida que el índice esté fuera de rango, la función miembro at sí lo hace.

- El tipo de errores del segundo caso es más fácil de detectar pues, con frecuencia, la cadena aparentemente permanece sin cambios aun después de haber realizado una operación de reemplazo o de haber borrado sobre ella. Esto debe conducir al lector a revisar la correcta asignación del resultado devuelto por las funciones miembro que forman parte del programa.

6.3 Uso de cadenas de caracteres para procesar textos

El procesamiento de cadenas de caracteres persigue muy diversos fines, desde lo más simple como extraer un dato, hasta lo más complejo como generar esquemas de análisis de información. Una de las áreas más importantes y conocidas que está relacionada con este concepto es la **criptografía** la cual tiene varios fines uno de ellos es la confidencialidad. Este objetivo plantea el reto de tomar un mensaje y modificarlo de tal forma que sea ininteligible para un potencial intruso que lo intercepte, pero que al mismo tiempo se transforme de manera inversa, convertirse en el mensaje original, al llegar a su destino. A lo largo de la historia de las ciencias computacionales se han diseñado muchos y muy complejos algoritmos para lograr este objetivo. En la Figura 6.4 podrá observar una propuesta de solución muy simple, pero ilustrativa.

Problema

Elabore un programa de codificación que pida por teclado el texto que se quiere cifrar y una contraseña para cifrarlo. El programa deberá mostrar el texto cifrado. Enseguida el programa deberá aplicar el algoritmo inverso y descifrar el mensaje original usando una contraseña proporcionada por teclado. El algoritmo debe alterar la distribución de las letras del alfabeto, es decir, no debe cifrar de la misma forma siempre los mismos caracteres.

Análisis del problema

Datos de entrada:

- El mensaje a codificar y la contraseña de codificación.

Datos de salida:

- El mensaje cifrado.
- El mensaje descifrado.

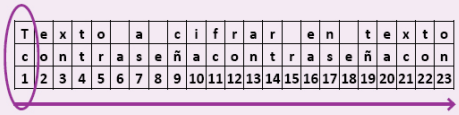
Planteamiento del problema:

Dados un mensaje y una contraseña de cifrado, elaborar un algoritmo de cifrado y descifrado basados en dicha clave.

Diseño del algoritmo

Consideraciones de diseño:

- Para romper la distribución de las letras del alfabeto se tiene que cifrar con base en la contraseña de tal forma que la posición del caracter afecte su cifrado.
- Pueden colocarse el texto a cifrar, la contraseña y el índice a la par. Luego repetirse la contraseña tantas veces como sea necesario para cubrir todo el mensaje.
  - Basta con alterar el caracter original usando el código ASCII del caracter correspondiente en la contraseña afectado por el índice.



Diseño del algoritmo

```
Algoritmo codificacion
Variables
    string texto, clave, cifrado, descifrado;
Inicio
    Escibir "Texto a cifrar: "        Leer texto
    Escribir "Clave de cifrado: "    Leer clave
    cifrado = cifrar(texto, clave)
    Escribir "Texto cifrado: ", cifrado

    Escribir "Clave de cifrado: "    Leer clave
    descifrado = descifrar(cifrado, clave)
    Escribir "Texto descifrado: ", descifrado
Fin
```

Diseño del algoritmo

```
Función cifrar( texto, clave)
Variables
    string cifrado;
    int indice, longitud;
    char letraTexto, letraClave, nuevaLetra;
Inicio
    longitud ← texto.longitud()
    Para indice←0 Hasta longitud con Paso 1 Hacer
        letraTexto ← texto.en(indice)
        letraClave ← clave.en(indice mod clave.llongitud())
        nuevaLetra ← (letraTexto + letraClave*(indice+1)) mod 256;
        cifrado ← cifrado + nuevaLetra;
    Finpara
    Devolver cifrado
Fin
```

Diseño del algoritmo

```
Función descifrar( texto, clave)
Variables
    string descifrado;
    int indice, longitud;
    char letraCifrado, letraClave, nuevaLetra;
Inicio
    longitud ← texto.longitud()
    Para indice←0 Hasta longitud con Paso 1 Hacer
        letraCifrado ← texto.en(indice)
        letraClave ← clave.en(indice mod clave.longitud())
        nuevaLetra ← (letraCifrado -- letraClave*(indice+1)) mod 256;
        descifrado ← descifrado + nuevaLetra;
    Finpara
    Devolver descifrado
Fin
```

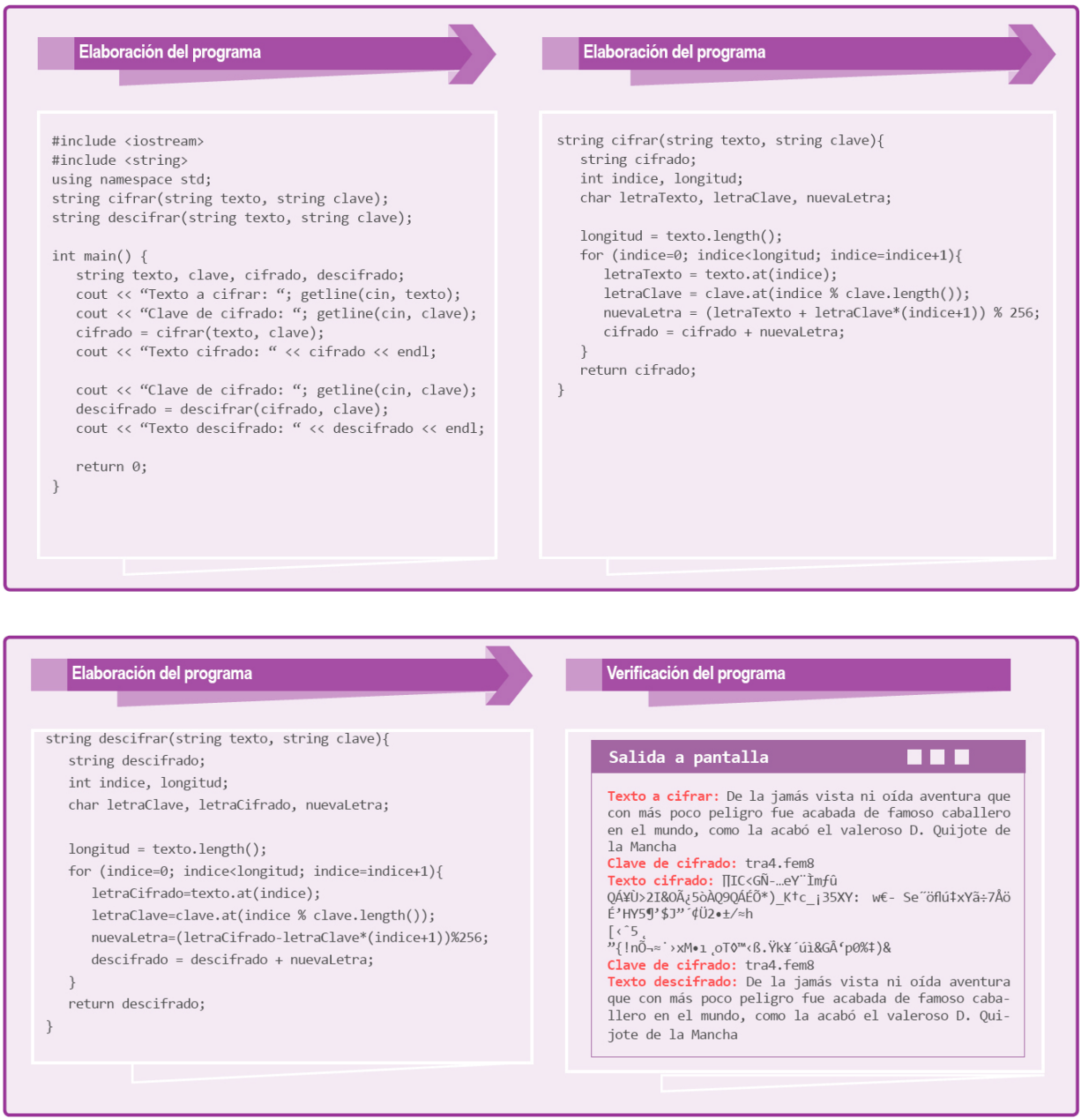


Figura 6. 4. Solución al problema de la codificación.

Las transformaciones que se observan en el problema de la Figura 6.4 son una muestra clara de la versatilidad de los lenguajes de programación. En la actualidad, se utilizan algoritmos más complejos que realizan transformaciones más elaboradas en millones de operaciones financieras al día. Estas operaciones, por lo general, no solamente actúan sobre texto sino sobre todo tipo de

información, desde números de cuenta e imágenes hasta elementos tridimensionales completos.

6.4 Concatenación y división, delimitadores

Una de las necesidades más comunes en relación al procesamiento de cadenas de caracteres consiste en la separación, análisis y clasificación de los componentes de las cadenas de texto; normalmente estas se dividen en subcadenas, palabras o frases de acuerdo con parámetros específicos.

El lenguaje natural, por ejemplo, se divide en palabras para posteriormente identificar elementos como sustantivos y verbos, entre otros. La información numérica se divide en cifras, dígitos y operadores. Sin importar cuál sea el fin último del análisis, a este procedimiento inicial se le conoce como descomposición o **parsing**. Este procedimiento se enfoca en dividir el texto en componentes simples más fáciles de procesar y analizar que el texto en su conjunto.

Esta división se lleva a cabo con base en **delimitadores**, que son símbolos con un significado específico, en el lenguaje natural uno de ellos podría ser el punto final de las oraciones; en los lenguajes de programación uno de ellos es el punto y coma que se colocan al final de una línea. En la Figura 6.5 podrá identificar el manejo de diferentes funciones en la resolución de un problema específico de descomposición.

Problema

Elabore un programa que reciba del usuario una cadena de texto que representa el dominio de un sitio web. El programa en cuestión deberá procesar la cadena de texto para devolver el dominio invertido, el cual es útil para identificar la jerarquía y rutas asociadas con dichas estructuras.
Por ejemplo para profesional.cva.itesm.mx deberá devolver mx.itesm.cva.profesional

Análisis del problema

Datos de entrada:

- El nombre del dominio

Datos de salida:

- El dominio invertido

Planteamiento del problema:

Dada una cadena que representa el dominio de un sitio web, obtener el dominio invertido en función de los separadores

Diseño del algoritmo

Consideraciones de diseño:

- El caracter delimitador es el punto

- Se necesita identificar el primer punto
- Enseguida obtener la sub cadena del inicio
- Insertar dicha sub cadena al inicio del dominio inverso
- Repetir la operación a partir del último punto encontrado

profesional.cva.itesm.mx	⟵ ⟶
cva.itesm.mx	⟵ ⟶ profesional
itesm.mx	⟵ ⟶ cva.profesional
mx	⟵ ⟶ itesm.cva.profesional
	⟵ ⟶ mx.itesm.cva.profesional

Diseño del algoritmo

```
Algoritmo dominio
Variables
    string dominio, invertido, segmento
    int inicio, fin
Inicio
    Escribir "Nombre de dominio: "
    Leer dominio
    inicio ⟵ 0
    Mientras ( (fin ⟵ dominio.buscar(".", inicio)) >= 0) Hacer
        segmento ⟵ dominio.subcadena(inicio, fin – inicio)
        invertido ⟵ "." + segmento + invertido
        inicio ⟵ fin + 1
    Finmientras
    invertido ⟵ dominio.subcadena(inicio, dominio.longitud() – inicio)
                + invertido
    Escribir "Nombre de dominio invertido: ", invertido
Fin
```

Elaboración del programa

```
#include <iostream>
#include <string>
using namespace std;

int main() {
   string dominio, invertido, segmento;
   int inicio, fin;

   cout << "Nombre de dominio: ";  getline (cin, dominio);
   inicio = 0;
   while ( (fin = dominio.find(".", inicio)) >= 0){
      segmento = dominio.substr(inicio,fin - inicio);
      invertido = "." + segmento + invertido;
      inicio = fin+1;
   }
      invertido  =  dominio.substr(inicio,  dominio.length()-
inicio)+invertido;
   cout << "Nombre de dominio invertido: " << invertido << endl;
   return 0;
}
```

Verificación del programa

Salida a pantalla

```
Nombre de dominio: profesional.cva.itesm.mx
Nombre de dominio invertido: mx.itesm.cva.
profesional
```

Figura 6. 5. Solución al problema de identificación de dominio.

Como se observa en el ejemplo de la Figura 6.5, la separación de una cadena de caracteres en subcadenas puede tener diferentes propósitos de acuerdo con el problema específico por resolver. A

estas subcadenas, extraídas de la cadena de caracteres original, se les conoce más comúnmente por su término en inglés: *token.*

Las cadenas de caracteres constituyen el vasto universo de información que existe. Áreas completas de las ciencias computacionales se han dedicado por años al análisis y estudio de esta información. Términos como **web mining**, **text mining** e **image mining** son ejemplos de algunas de estas áreas.

Conforme cada una de estas áreas alcance un grado de madurez suficiente, se observarán sus frutos en los dispositivos de uso cotidiano, tal como ha sucedido ya con los avances tecnológicos de los que disfruta el hombre hoy en día.

6.5 Piense en grande: caso Watson, el nuevo paso en la evolución computacional

Como se ha visto en este capítulo, el análisis de la información que contienen cadenas de caracteres es un punto de partida imperativo para el procesamiento automático de la información.

La pregunta es: ¿hasta qué punto puede una computadora hacer un análisis de lo que procesa?

Ante esta interrogante desarrolladores de IBM han generado, en los últimos años, un proyecto llamado *Watson* que cumplió, en marzo de 2011, la difícil tarea de vencer a los campeones mundiales del juego *Jeopardy.* Además del análisis del lenguaje natural y de la inteligencia artificial que se emplearon, cabe recalcar la complejidad de la interpretación de las preguntas de este concurso, ya que muchas de ellas incluyen figuras retóricas como el símil o la metáfora. Observe cuidadosamente de la barra lateral para conocer un poco más acerca de este interesantísimo proyecto.

Como puede apreciarse en el video, ubicado en la barra lateral, aunque el procesamiento es complejo y los algoritmos no son

triviales, todo el proyecto se genera a partir de la intención de emular las capacidades del ser humano, no con la intención de sustituirlo sino para que la computadora lo asista en la toma de decisiones importantes. En la actualidad son muchas ya las áreas en las que esta tecnología tiene un gran impacto, desde el área de la salud, en el diagnóstico, en el ámbito financiero, al hacer análisis comparativos, hasta en el apoyo al desarrollo de ciertas habilidades humanas como una asesoría virtual.

Actividad de repaso

Problema 1. Cuenta palabras

Elabore un programa que pida al usuario escribir una frase, enseguida deberá pedirle que escriba una palabra. El programa ha de determinar cuántas veces aparece la palabra completa dentro de la frase sin importar si está al inicio, al final o en medio del enunciado. Ejemplo de la ejecución: (los datos subrayados son los que se capturan por teclado)

```
Frase: la regla labrada de la que canta en la
Palabra: la
La palabra aparece: 3 veces
```

Problema 2. Extraer números

Una empresa de telefonía necesita equipar a sus teléfonos celulares con la capacidad de extraer números de los mensajes de texto. Elabore un programa que pida al usuario que capture una frase. Enseguida el programa deberá extraer todos los números que aparezcan dentro de la frase y mostrarlos en pantalla.

Ejemplo de ejecución:

```
Indique una frase: Hola soy Rodolfo, mi mail
es rodo@ hotmail.com, llámame al 7773400021 a
las 5 p.m. necesito 60 piezas más de tipo R45

Los números que contiene el mensaje son:

7773400021

5

60
```

Problema 3. Cálculo del RFC

Elabore un programa que calcule automáticamente el RFC de acuerdo con las siguientes reglas. El programa debe calcular el RFC de 10 caracteres, sin homoclave.

Reglas:

1. Se toma la primera letra del apellido paterno y la primera vocal después de la primera letra.
2. Se toma la primera letra del apellido materno.
3. Se toma la primera letra del nombre.
4. No se toman en cuenta los artículos y preposiciones “de”, “del”, “la”, “los”, “las”, “y”.
5. Si tiene más de un nombre y empieza con jose el “jose” no se toma en cuenta.
6. Si tiene más de un nombre y empieza con maria, el “maria” no se toma en cuenta.
7. Se toman los dos últimos dígitos del año de nacimiento.
8. Se toman los dos dígitos del mes de nacimiento.
9. Se toman los dos dígitos del día de nacimiento.

Existen más reglas, pero con que cumpla las anteriores es suficiente. Haga pruebas con el RFC de personas que conozca. Deben introducirse los nombres todos en minúsculas y sin acentos.

Problema 4. Juego del ahorcado

Elabore un programa que juegue con el usuario el juego de “el ahorcado”, el juego consiste en lo siguiente: el programa debe considerar una palabra cualquiera, no se la debe mostrar al usuario, sólo le debe indicar de cuántas letras es. En cada oportunidad el usuario indica una letra, si la letra se está dentro de la palabra se le muestra al usuario las posiciones donde aparece, de otro modo el usuario ha perdido una oportunidad y se le agrega, al dibujo de un muñeco en una horca, una parte

del cuerpo. El usuario gana si descubre todas las letras de la palabra y pierde si el muñeco se dibuja completamente.

Problema 5. Sopa de letras

Existe un juego llamado Sopa de letras que trata de localizar, dentro de una matriz de letras, una palabra conocida con anterioridad.

Dada una matriz de caracteres, elabore un programa que pida al usuario escribir una palabra con el teclado. Enseguida, el programa deberá determinar si la palabra existe o no dentro de la sopa de letras y en qué coordenadas está ubicada.

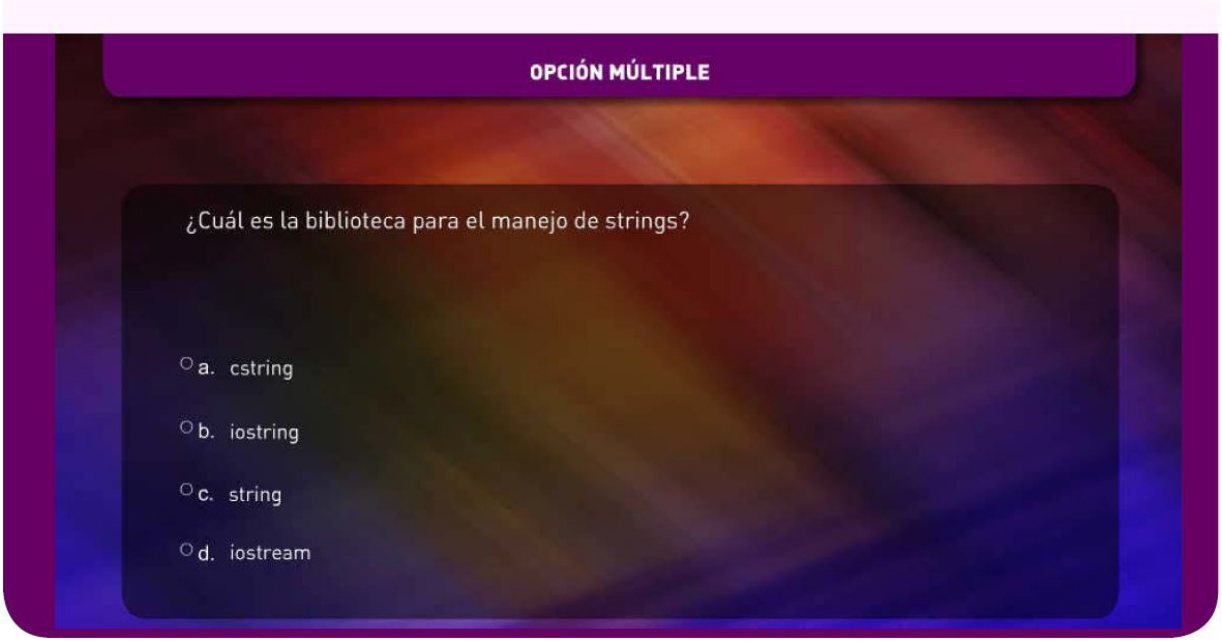
Problema 6. Matriz de eventos

En el Tecnológico de Monterrey hay eventos, como los ciclos de conferencias, que se efectúan en el transcurso de un día completo. Para llevar un mayor orden se requiere organizar las conferencias en relación con los espacios físicos disponibles. Elabore el programa que me permita realizar una asignación manual de conferencias en los distintos salones para una fecha específica.

Para ello debe utilizar una matriz. Cada fila corresponderá a un periodo de tiempo; es decir, la primera fila serán los eventos de 7 a 8 a.m.; la segunda fila, los de 8 a 9 a.m.; y así sucesivamente. Las columnas corresponderán con los salones; en principio hay ocho salones disponibles.

El programa deberá preguntarle al usuario cuál es el nombre del evento, la hora de inicio y la duración en horas. El programa deberá reservarle el espacio en caso de que la hora de inicio esté libre y cuidará que no se traslape con ninguna actividad asignada anteriormente. En caso de haber traslape, hay que indicárselo al usuario. También debe indicarle si el número de horas excede el horario programado de eventos: de 7 a.m. a 6 p.m. Por ejemplo, si se quiere programar una actividad a las 5 p.m. de tres horas de duración, el programa debe indicar que eso no es posible.

Ejercicio integrador del capítulo 6



Conclusión del capítulo 6

Imagine por un momento la cantidad y variedad de idiomas y dialectos que se hablan en el mundo, además de las reglas sintácticas y semánticas que los rigen. Aunado a la complejidad de los idiomas, el volumen de información que se genera crece día con día. A diferencia de siglos anteriores, en el nuestro resulta prácticamente imposible reunir el conocimiento generado, mucho menos obtener conclusiones valiosas de esta información.

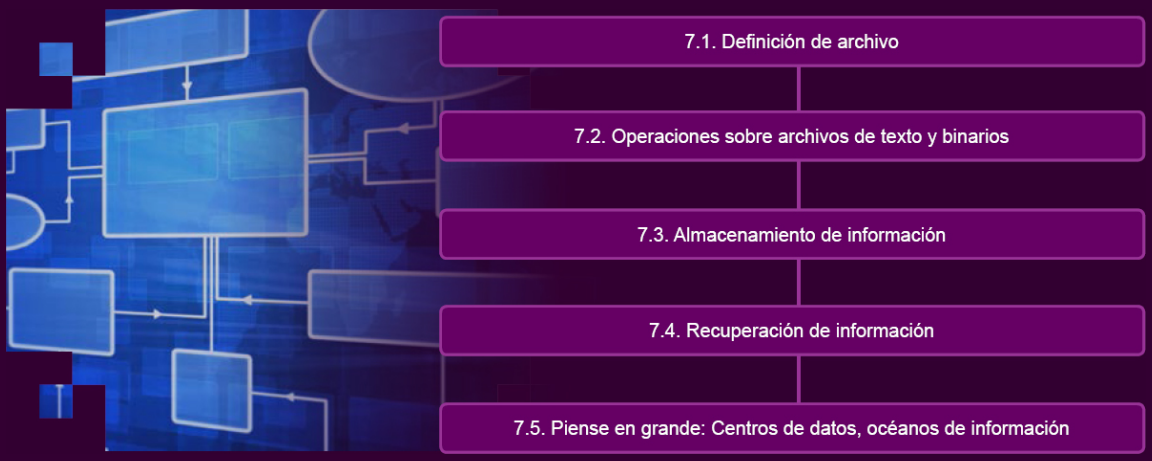
Llegado el momento será necesaria la ayuda de sofisticadas piezas de software como Watson para desenmarañar la telaraña ininteligible en la que se habrá vuelto el conocimiento. Hoy mismo se comienza a vislumbrar la necesidad de dar el siguiente paso en lo referente al análisis e interpretación de información.

Hace algunos años los buscadores de internet resultaron un valioso apoyo en la exploración de información relevante. Ahora, al introducir un término en un buscador muchas veces se obtiene una lista de doscientos mil resultados, unos valiosos y otros no tanto.

Generar conocimiento útil a partir de esta información, clasificarla automáticamente y sobre todo filtrar contenido relevante, son tareas aún pendientes; algunas de éstas pronto serán resueltas, al menos parcialmente; otras apenas empiezan a ser imaginadas.

Capítulo 7. Archivos: almacenamiento persistente

Organizador temático



Archivos: almacenamiento persistente

7.1 Definición de archivo

El almacenamiento persistente de la información es un aspecto relevante en el funcionamiento de los programas que se utilizan cotidianamente. La gran mayoría de los procesos que se ejecutan en la computadora arrojan resultados factibles de ser almacenados: archivos, imágenes, números y datos.

En la actualidad, el almacenamiento se ha movido hacia un esquema distribuido en el que la información ya no se guarda sólo en un disco duro local. Se almacena en un espacio virtual accesible

por medio de internet con la finalidad de acceder a la información desde múltiples puntos y de conservar, al mismo tiempo, la confidencialidad de la misma.

A este esquema de almacenaje se le conoce como **almacenamiento en la nube** y plantea tanto ventajas como desventajas. Por un lado permite a las personas tener facilidad de acceso a la información que se guarda; en contraparte, esta información se almacena en el sistema de archivos de un **servidor** conectado a internet, por lo que una violación física o remota a este servidor pondría en riesgo la confidencialidad de la información. Mucho se discute actualmente al respecto, por lo que se espera que estos esquemas evolucionen en productos que logren un equilibrio entre el fácil acceso y la confidencialidad de los datos que se resguardan.

Sin importar si el almacenamiento es remoto o local, los mecanismos para su gestión son esencialmente los mismos; en ellos el archivo es la unidad mínima básica de almacenamiento.

Es archivo se define como un repositorio de información almacenada de manera persistente.

Un archivo puede tener diferente información y características dependiendo de si almacena una imagen, un correo electrónico, un texto o un modelo en 3D. Básicamente hay dos maneras principales de identificar el tipo de información que contiene un archivo y, en consecuencia, la aplicación que puede utilizarse para procesarlo:

1. La primera es por medio de un código específico colocado al inicio del archivo al interior de él. Este mecanismo emplea unos cuantos bytes al inicio del archivo como indicativo de que pertenece a cierto tipo.

2. La segunda es basándose en la extensión del archivo, es decir, las letras que siguen al nombre del archivo después del punto.

Según lo crítico de la identificación, uno u otro método pueden ser utilizados para diferentes aplicaciones tal y como lo hacen los editores de texto, los editores de imágenes y muchas otras herramientas.

En C++ el acceso a archivos, ya sea para almacenar o recuperar información, se lleva a cabo por medio de **flujos** o streams. Esta abstracción permite aprovechar los mecanismos ya vistos en el capítulo 6 para el manejo de streams. Por supuesto, hay instrucciones adicionales para su manejo, pero una vez abierto, un archivo funciona básicamente como los flujos cin y cout; para la recuperación y almacenamiento de información respectivamente. En la Tabla 7.1 podrá identificar algunas de las principales funciones miembro que se utilizan en el manejo de archivos.

Tabla 7.1. Principales funciones miembro para el manejo de archivos.

Biblioteca	Función	Descripción
<fstream>	`void open(char *filename, openmode mode);`	Permite abrir un archivo. A esta función se le proporciona como primer parámetro el nombre del archivo, y como segundo parámetro los modificadores correspondientes al modo de apertura.
	`void close();`	Permite cerrar un archivo. Opera sobre el identificador que hace el llamado a la función miembro.
	`basic_istream& read(char* s, size count);`	Permite leer información de un archivo. Se debe especificar como primer parámetro una dirección de memoria en la que se coloca la información leída, y

		como segundo parámetro el número de bytes a leer.
	`basic_ostream& write(char* s, size count);`	Permite escribir información en un archivo. Se debe especificar como primer parámetro una dirección de memoria de donde se tomará la información, y como segundo parámetro el número de bytes a escribir.

De manera adicional, hay dos formatos base en los que C++ puede almacenar información: el *modo texto* y el *modo binario.* La diferencia entre uno y otro es significativa. Para identificar las principales diferencias entre un formato y otro observe la información de la Figura 7.1.

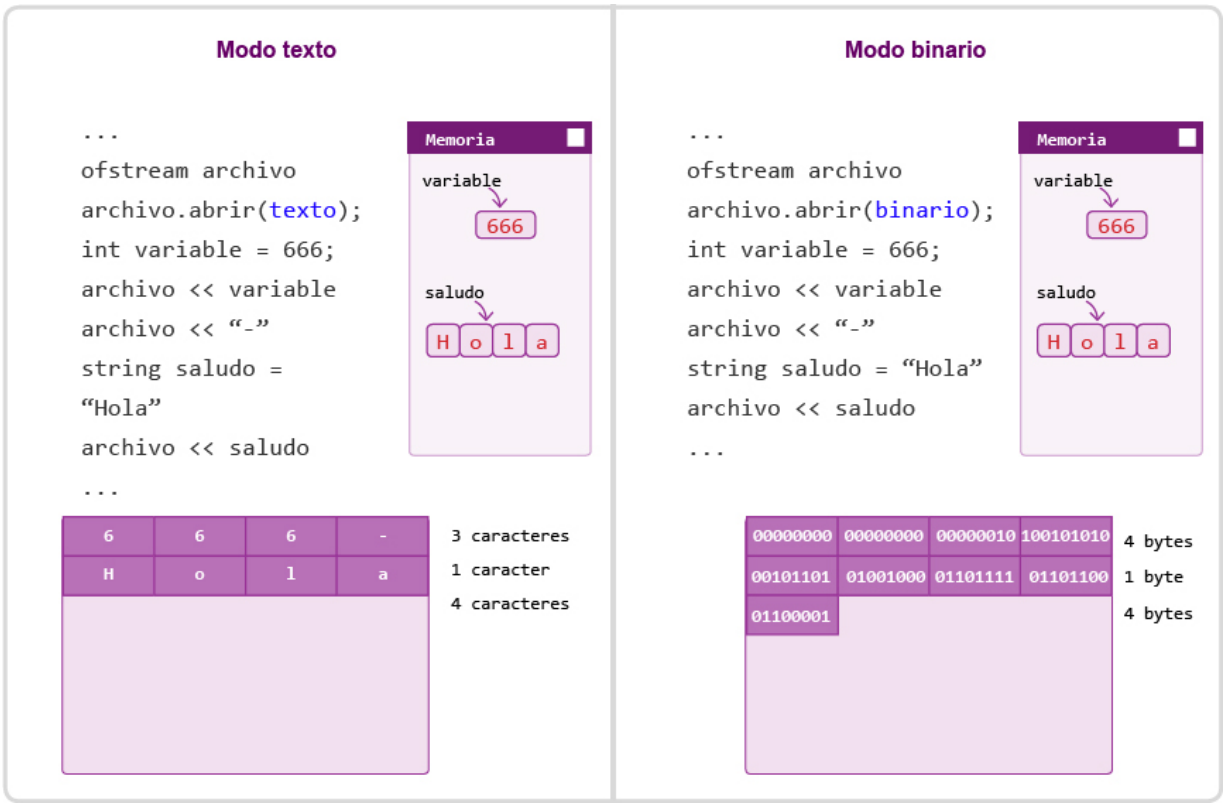


Figura 7.1. Formatos base para el almacenamiento de información en archivos.

Como se puede identificar en la Figura 7.1 mientras los archivos en modo texto son legibles a simple vista mediante un editor de texto, los archivos binarios no ofrecen información de primera mano, almacenan bytes tal y como están en la memoria, en código binario, por lo que es necesario saber la forma en que está almacenada la información para luego procesarla e interpretarla.

Ninguno de los dos mecanismos es mejor que el otro simplemente se trata de formatos diferentes. La decisión de cuál utilizar radica en las características de la información que será almacenada y en el propósito de su almacenamiento. Factores como la confidencialidad podrían llevar a decidirse por el modo binario, mientras que otros como la facilidad de acceso podrían llevar a optar por el modo de texto.

7.2 Operaciones sobre archivos de texto y binarios

Una vez que se ha planteado el concepto de almacenamiento y la diferencia entre los dos principales modos, toca el turno a identificar la sintaxis que debe emplearse en cada caso. Si se comienza por el manejo de archivos en modo texto, se requiere la inclusión de la biblioteca *fstream,* cuyo nombre se deriva del término *File Stream.* En la Figura 7.2 podrá identificar las principales instrucciones utilizadas en este modo de almacenamiento.

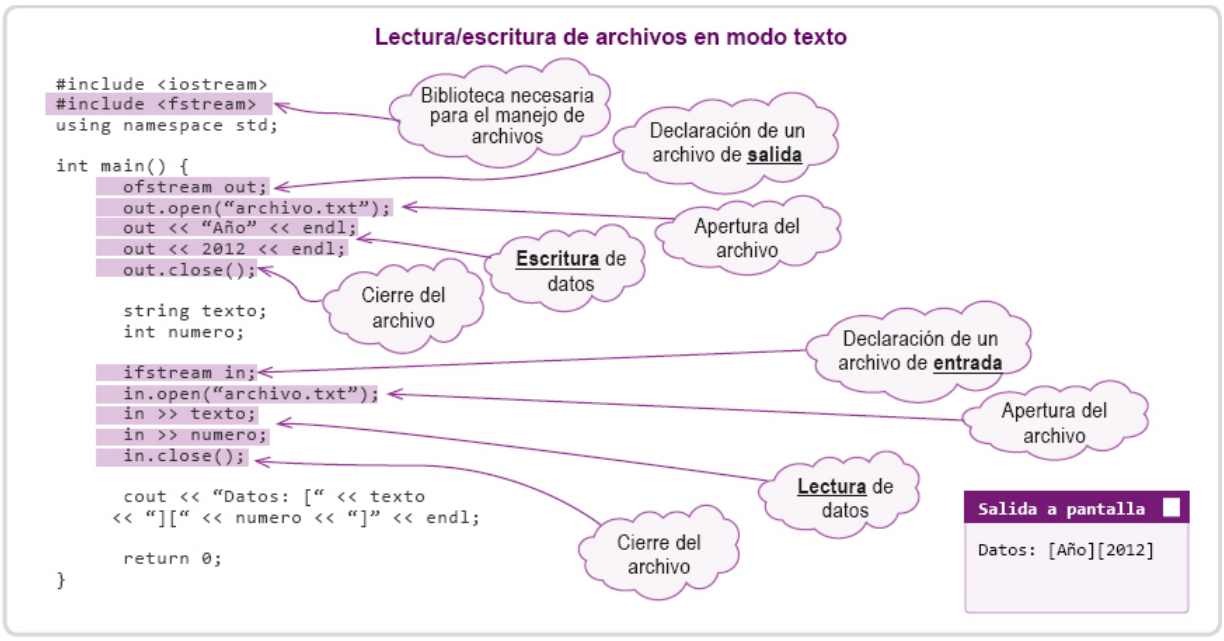


Figura 7. 2. Instrucciones para el manejo de archivos en modo texto.

Un punto muy importante del almacenamiento en modo texto, visto en la Figura 7.2, es la diferencia entre la lectura de datos sin formato o con formato, es decir, la diferencia entre la utilización de la función *getline* sobre el flujo que representa un archivo o su procesamiento por medio del operador representado por >> o <<, en combinación con tipos de datos específicos. Del ejemplo anterior es

interesante observar que el archivo generado se reporta por el sistema como un archivo de nueve bytes, es decir: los tres bytes de la palabra “Año” + el caracter de salto de línea (que también de envía al flujo) + 4 bytes del entero “2012” + el caracter de salto de línea. Lo cual da un total de exactamente 9 bytes.

Por otro lado, el procesamiento de archivos en formato binario se lleva a cabo mediante la misma biblioteca fstream y con las mismas instrucciones de apertura y cierre. Para habilitar el manejo de archivos en modo binario se usa el modificador *binary* definido en la biblioteca iostream como parámetro al abrir un archivo. Para identificar las principales instrucciones utilizadas en el modo de almacenamiento binario revise la Figura 7.3.

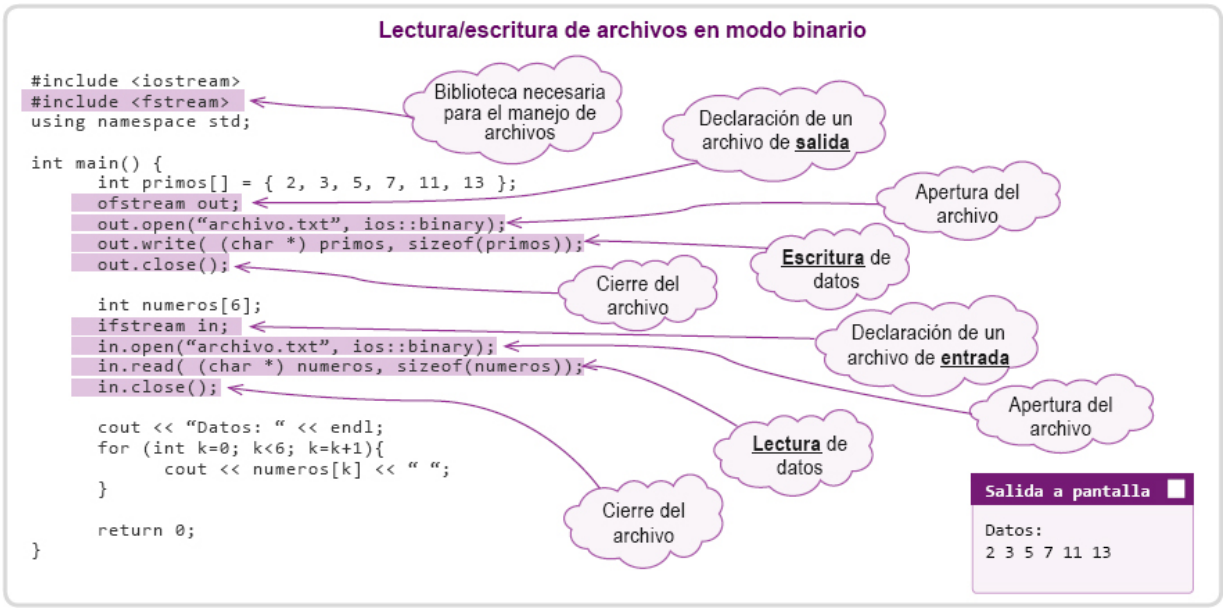


Figura 7. 3. Instrucciones para el manejo de archivos en modo binario.

Como puede observarse en el ejemplo, el acceso a la información se ejecuta mediante instrucciones *read* y *write,* las cuales operan a nivel de bytes directamente sobre direcciones de memoria. En el ejemplo se muestra cómo puede almacenarse un arreglo completo con base en su referencia.

Es importante hacer notar que de acuerdo con las firmas de las funciones miembro read y write se pide siempre un elemento de tipo

(char *) como primer parámetro. Esto se debe a que la información se procesa en múltiplos de un byte, no se refiere a que deba tratarse de una cadena de caracteres ni mucho menos. Basta con hacer un casting explícito independientemente del tipo del que se trate. Lo que sí resulta de vital importancia es que en el segundo parámetro se indique correctamente el número de bytes que ocupa la variable en cuestión. Esto último puede obtenerse por medio de la función *sizeof.*

Nuevamente cabe recalcar que el archivo generado se reporta por el sistema como un archivo de veinticuatro bytes, es decir, cuatro bytes por cada entero, multiplicado por seis que es el número de elementos del arreglo. Observe como en esta representación no hay saltos de línea ni separaciones entre los valores, ya que todos se almacenan de forma continua.

Los datos se almacenan en su representación binaria, simple y llanamente de la manera en que están guardados en la memoria sólo que dirigidos hacia el flujo que representa el archivo, sin seguir ningún formato susceptible de interpretación.

7.3 Almacenamiento de información

El almacenamiento de información representa la mitad de la ecuación en lo que al manejo de archivos de refiere. De manera consistente con las secciones anteriores del capítulo puede esperarse un manejo radicalmente diferente entre los archivos en modo texto con respecto de los archivos en modo binario. En primer lugar, revise la Figura 7.4 para identificar las diferentes maneras en que puede abrirse un archivo usando la función miembro llamada *open.*

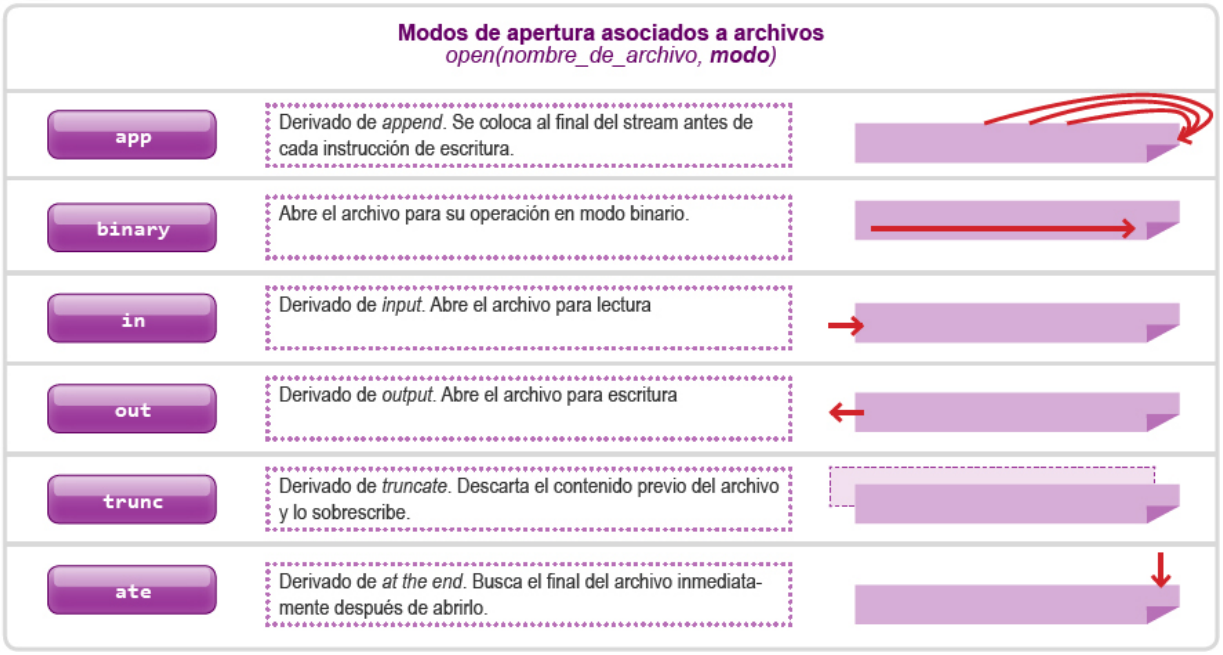


Figura 7. 4. Modos de apertura de un archivo.

A lo que se presenta en la Figura 7.4 debe sumarse el hecho de que estos valores están contenidos en ios_base, lo cual debe indicarse de manera explícita para poder usar cualquiera de ellos; por ejemplo, en el caso de binary tendría que colocarse *ios_base::binary.* Si se diera el caso de que se requieren usar dos o más de estos modos de apertura, basta con hacer un OR lógico entre ellos para incorporarlos ya que se trata de máscaras de bits. Por ejemplo, si se desea usar binary en combinación con ate, se deberá escribir *ios_base::binary* | *ios_base::ate.*

Otro aspecto muy importante, y en ocasiones subestimado, en el manejo de archivos radica en el correcto uso del estado del flujo. Dicho estado se actualiza automáticamente al momento de hacer operaciones de lectura y estructura. No atender los cambios en el estado de los flujos puede derivar en un mal funcionamiento de los programas y en inconsistencias a la hora de acceder a la información. En la Figura 7.5 podrá identificar las principales funciones asociadas con la verificación y manejo del estado de un flujo.

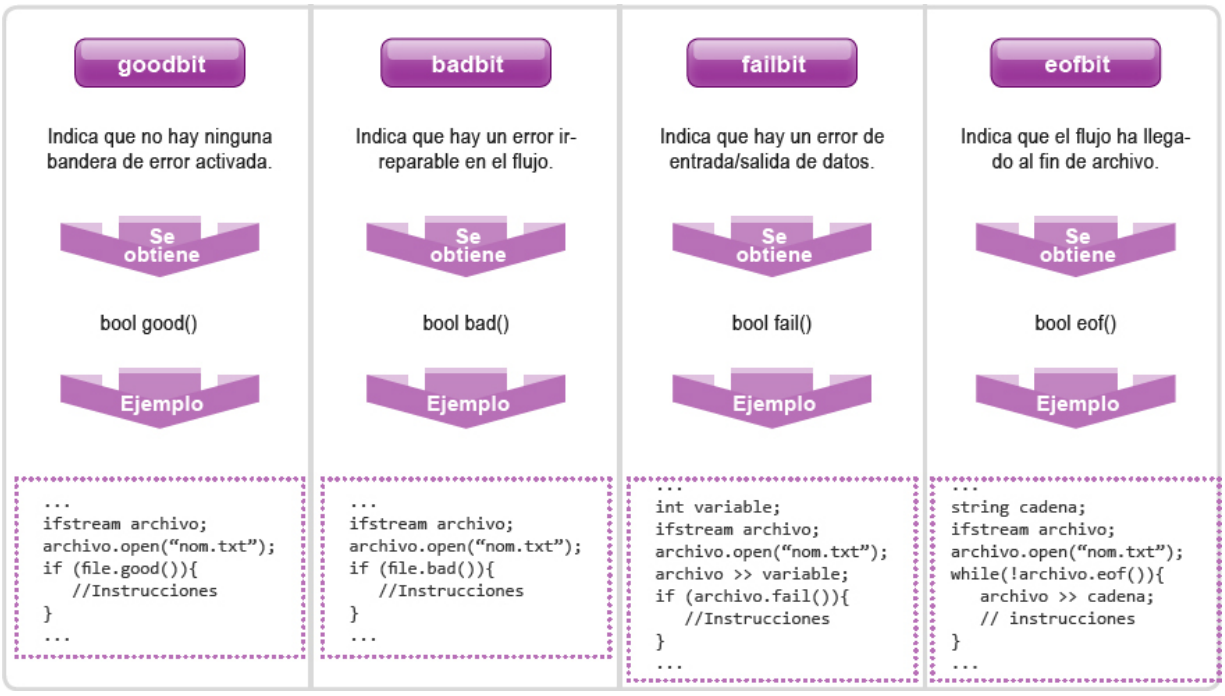
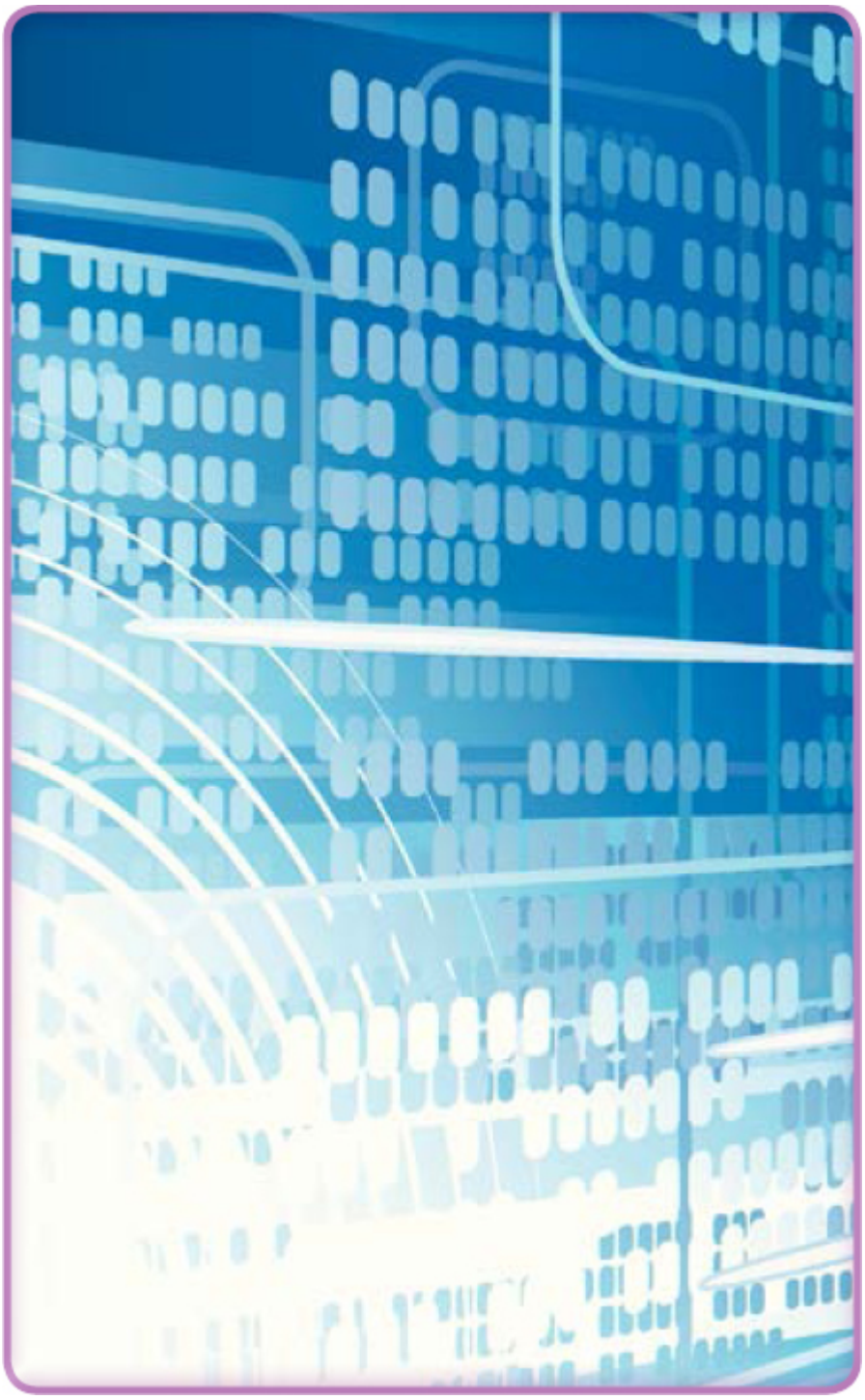
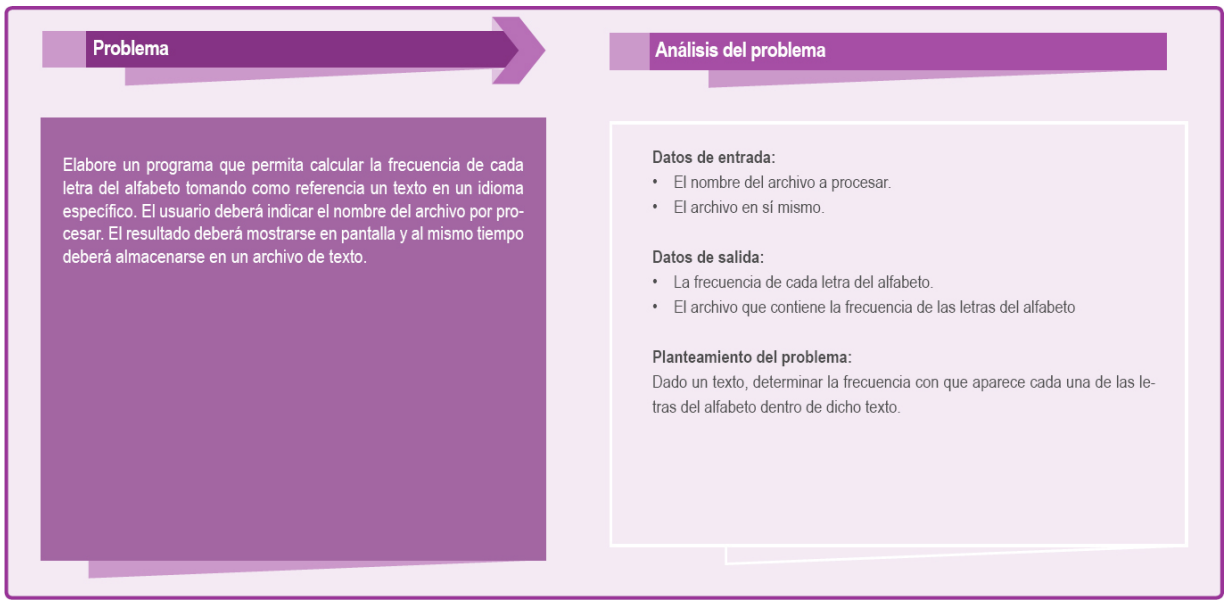


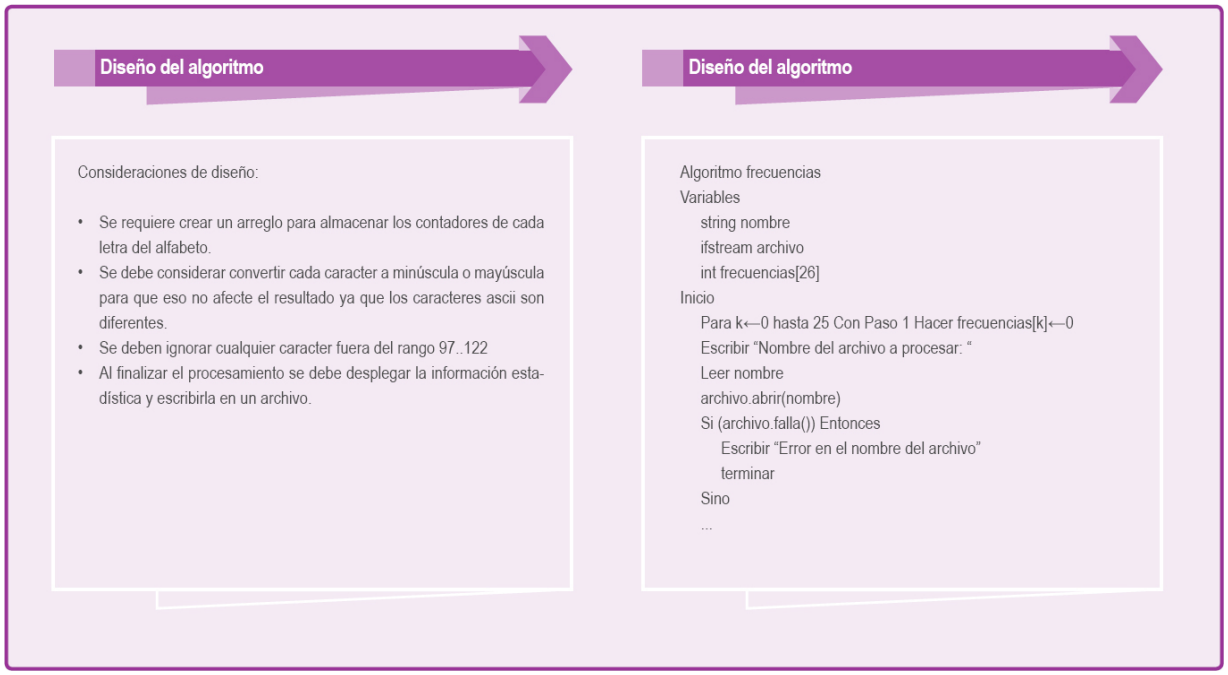
Figura 7. 5. Funciones para el manejo del estado de flujos.



Cabe mencionar que ninguna de las instrucciones que se han visto tendría sentido si no se aplican en la solución de un problema. Hay muchos ejemplos dado el enorme espectro de aplicaciones de la informática moderna. En la Figura 7.6 podrá identificar los pasos

en el análisis y resolución de un problema relacionado con el manejo de archivos.





Diseño del algoritmo

```
    calcularFrecuencias(frecuencias, archivo)
    archivo.cerrar()
    mostrarFrecuencias(frecuencias)

    return;
}

Función mostrarFrecuencias(frecuencias)
Variables
    k
Inicio
    Para k←0 Hasta 25 Con Paso 1 Hacer
        Escribir (char) (k+97)
        Escribir frecuencias[k]
Fin
```

Elaboración del programa

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;
void calcularFrecuencias(int[], ifstream&);
void mostrarFrecuencias(int[]);
int main() {
        string nombre;
        ifstream archivo;
        int frecuencias[26];

        for (int k=0; k<26; k=k+1) frecuencias[k]=0;
        cout << "Nombre del archivo a procesar: ";
        getline(cin, nombre);
        archivo.open(nombre.c_str());
        if (archivo.fail()){
                cout << "Error en el nombre del archivo" << endl;
                return 0;
        }
```

Elaboración del programa

```
    calcularFrecuencias(frecuencias, archivo);
    archivo.close();
    mostrarFrecuencias(frecuencias);
    return 0;
}
void calcularFrecuencias(int frecuencias[], ifstream &archivo){
    char letra;
    archivo >> letra;
    while(!archivo.eof()){
        letra = tolower(letra);
        if ( (int)letra >=97 && (int)letra <=122)
            frecuencias[ (int)letra - 97]++;
        archivo >> letra;
    }
}
void mostrarFrecuencias(int frecuencias[]){
   for (int k=0; k<26; k=k+1)
     cout << (char) (k+97) << ": " << frecuencias[k] << endl;
}
```

Verificación del programa

Salida a pantalla

```
Archivo a procesar: usher.poe.txt

a: 2601                 o: 2448
b: 442                  p: 649
c: 883                  q: 27
d: 1461                 r: 1993
e: 4136                 s: 1938
f: 942                  t: 2918
g: 603                  u: 1009
h: 2076                 v: 369
i: 2517                 w: 684
j: 18                   x: 58
k: 130                  y: 598
l: 1374                 z: 18
m: 805
n: 2394
```

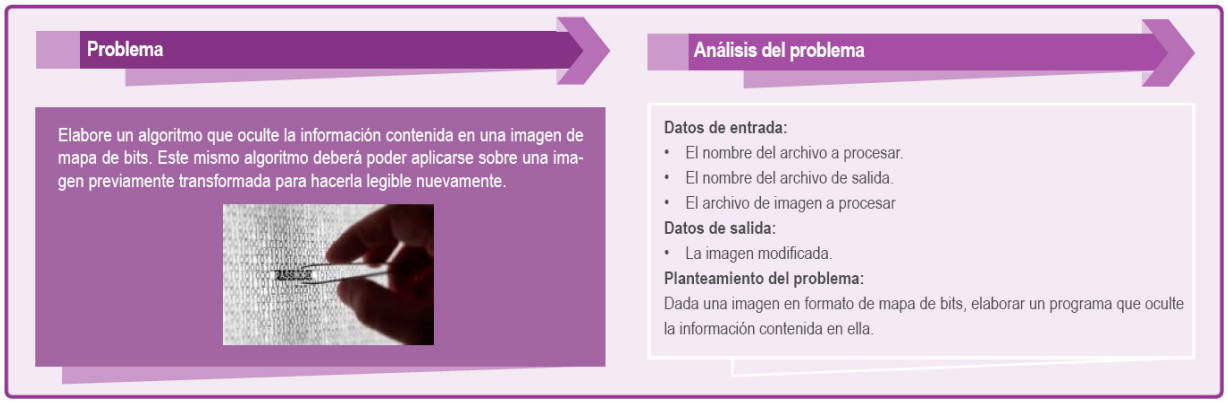
Figura 7. 6. Solución al problema del análisis de frecuencias.

Del problema que se presenta en la Figura 7.6 puede observarse el uso de la función miembro llamada *eof();* esta función devuelve verdadero cuando un flujo ha llegado al fin del archivo. Por otro lado, la función *tolower* de la biblioteca *cc-type* es utilizada para convertir un caracter a minúsculas. Un análisis estadístico como el que se

muestra en la figura se lleva a cabo cuando se intenta realizar un proceso denominado criptoanálisis que está orientado a identificar las transformaciones que se realizan por un algoritmo de cifrado sobre los caracteres del texto escrito en un idioma específico.

En el modo binario el punto medular es la correcta escritura de bytes en el archivo. Una vez que se tiene la dirección o referencia a una variable, arreglo o estructura, ésta se puede escribir completa en el archivo. Lo anterior es posible ya que en memoria la información no se almacena al usar delimitadores como las comas o los espacios; la información está almacenada secuencialmente en múltiplos de byte, dependiendo del tipo de dato específico.

En cuanto a las aplicaciones del almacenamiento de archivos en modo binario, el universo de posibilidades es mayor ya que facilita el manejo de estructuras complejas lo convierte en un formato ideal en muchos dominios. Por lo tanto, no resulta extraño que las imágenes, videos y aplicaciones que se utilizan día con día estén almacenadas en modo binario. En la Figura 7.7 identificará los pasos en el análisis y resolución de un problema relacionado con el manejo de archivos en modo binario.



Diseño del algoritmo

Consideraciones de diseño:

- Las imágenes de almacenan en archivos con formato binario.
- El formato de un mapa de bits indica una cabecera de 54 bytes la cual no debe modificarse ya que la imagen quedaría inservible o no podría ser identificada por el software de procesamiento de imágenes.
- Se puede usar el operador de bits XOR debido a que tiene el efecto reversible que se espera de la transformación.

Diseño del algoritmo

```
Igoritmo ocultarImagen
Variables
    string clave, nombreEntrada, nombreSalida
    int bytes
    ifstream entrada
    ofstream salida
Inicio
    Escribir "Imagen de entrada: "
    Leer nombreEntrada
    entrada.abrir(nombreEntrada, En Modo binario)
    Si (entrada.falla()) Entonces
        Escribir "Error al abrir archivo"
        Terminar
    Finsi
    Escribir "Imagen de salida : "
    Leer nombreSalida
    salida.abrir(nombreSalida, En Modo binario)
```

Diseño del algoritmo

```
    Escribir "Indique una contraseña: "
    Leer clave

    bytes ← convertirImagen(entrada, salida, clave)

    entrada.cerrar()
    salida.cerrar()
    Escribir "Se procesaron ", bytes," bytes "
Fin
```

Diseño del algoritmo

```
Función convertirImagen(entrada, salida, clave)
Variables
    byte, encabezado[54], indice
Inicio
    entrada.leer(encabezado, 54 bytes)
    salida.escribir(encabezado, 54 nytes)
    indice ← 0
    entrada.leer(byte, 1)
    Mientras (No entrada.finDeArchivo()) Hacer
        byte ← byte XOR
       (char) ( (indice * clave.en(indice Mod clave.longitud())) mod 256 )
        salida.escribir(byte, 1)
        entrada.leer(byte, 1)
        indice ← indice + 1
    Finmientras
    Devolver indice
Fin
```

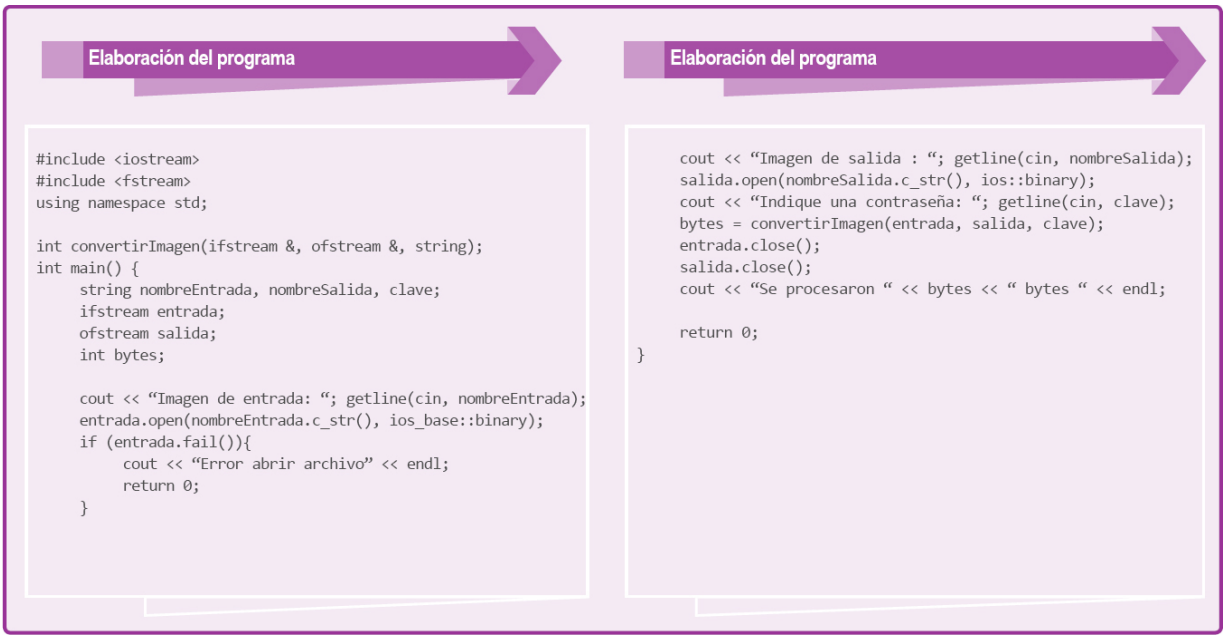
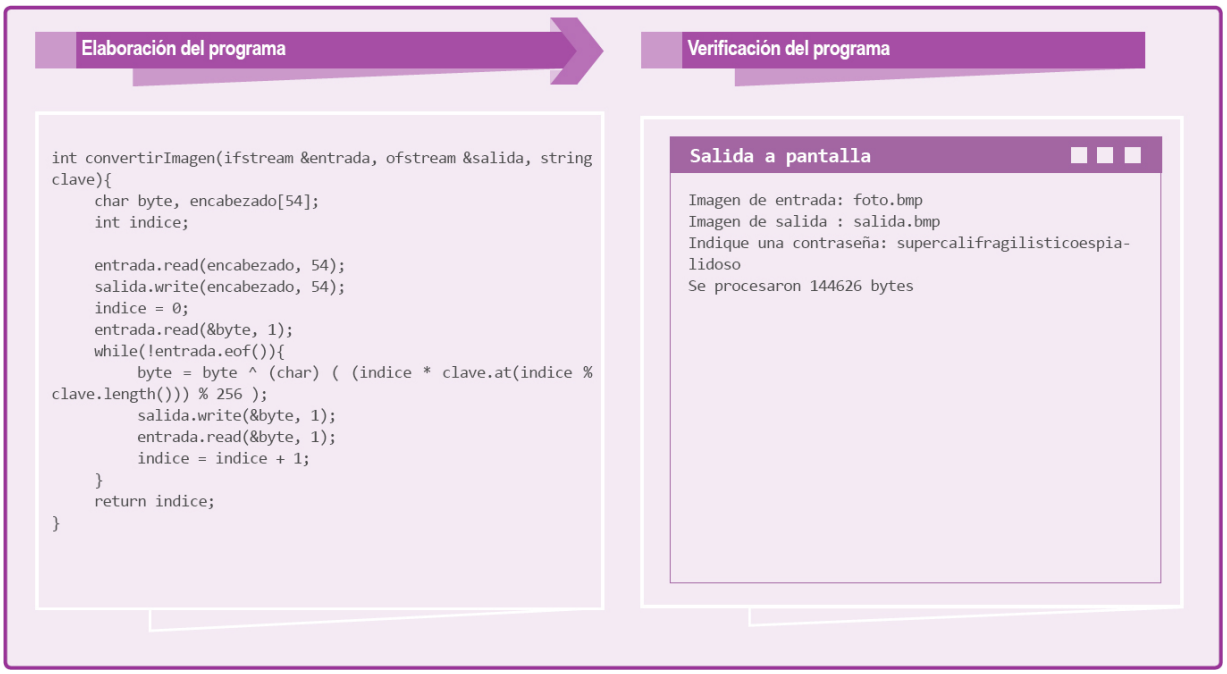


Figura 7. 7. Solución al problema de lectura de datos de una imagen.



En la Figura 7.8 se observa el resultado de aplicar la transformación a diferentes imágenes que se presenta en la Figura 7.7. Se usa el algoritmo descrito en dos ocasiones: la primera, para ocultar la información y la segunda, para recuperarla nuevamente.

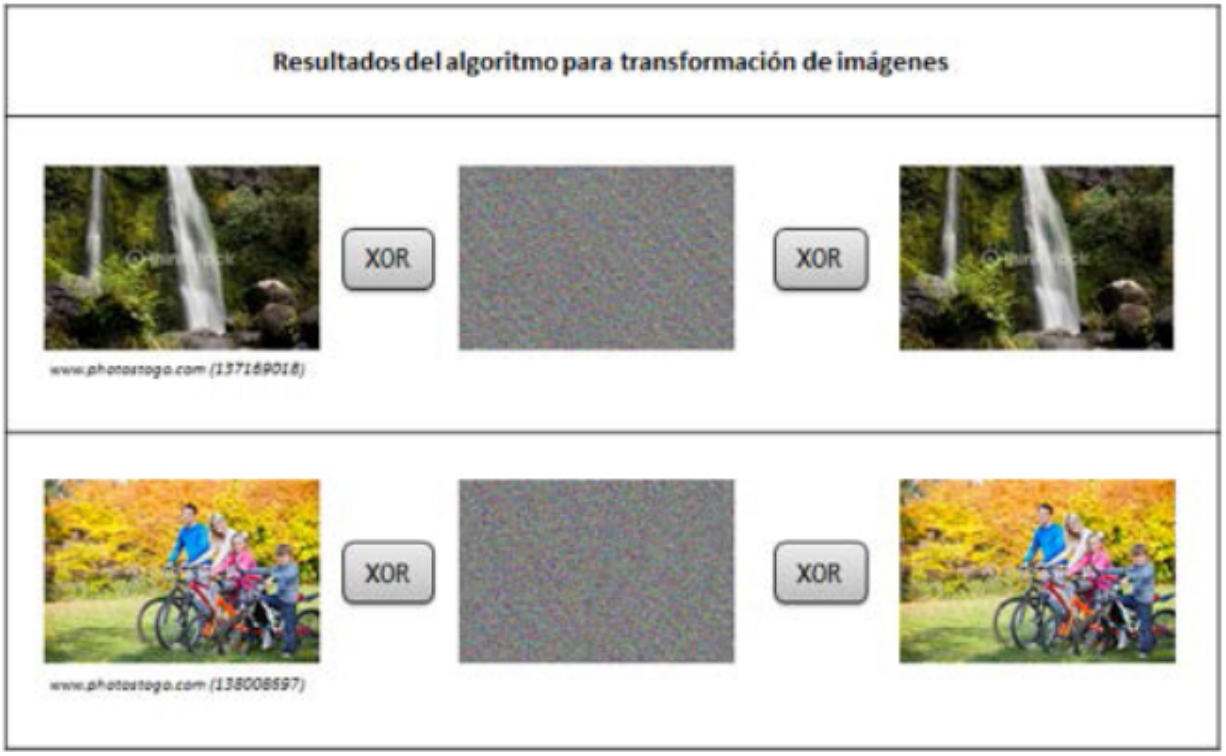


Figura 7. 8. Imágenes de ejemplo usando el operador XOR.

En contraparte al almacenamiento de la información está la recuperación de la misma. En la siguiente sección se analizan los detalles de la correcta recuperación de la información almacenada en cualquiera de los dos modos.

7.4 Recuperación de información

Cuando se quiere recuperar información de un archivo debe tomarse en cuenta no solamente el modo de almacenamiento que se empleó, sino también el propósito de su recuperación. Algunas veces solo se requiere recuperar un pequeño fragmento del archivo o sólo se necesita agregar información al final del mismo. Para identificar el uso de las instrucciones para el manejo de archivos en el contexto de la solución a un problema, revise la Figura 7.9.

Problema

Elabore un programa que permita calcular la frecuencia de cada letra del alfabeto tomando como referencia un texto en un idioma específico. El usuario deberá indicar el nombre del archivo por procesar. El resultado deberá mostrarse en pantalla y al mismo tiempo deberá almacenarse en un archivo de texto.

Análisis del problema

Datos de entrada:
- La distancia recorrida de cada prueba.
- La gasolina empleada en cada prueba.

Datos de salida:
- El resumen de los datos almacenados.
- El promedio de rendimiento de gasolina.

Planteamiento del problema:
Dado un archivo que contiene un conjunto de datos de rendimiento de un automóvil, mostrar el promedio de todos los casos y permitir agregar nuevos datos de pruebas en un archivo de texto.

Diseño del algoritmo

Consideraciones de diseño:

- Dado que se requiere almacenar un histórico de datos, se usa un archivo.
- Para agregar datos se puede hacer siempre al final del mismo usando el modificador de apertura de archivos correspondiente.
- Para obtener la información se puede leer por línea el archivo y obtener los datos correspondientes a la distancia y cantidad de gasolina empleada.

Diseño del algoritmo

```
Algoritmo rendimiento
Variables
        opcion
Inicio
        Hacer
                Escribir "  MENU PRINCIPAL "
                Escribir " 1. Agregar datos"
                Escribir " 2. Ver resumen"
                Escribir " 2. Salir"
                Escribir "Indique una opción: "
                Leer opcion
                Si (opcion = 1) Entonces agregarDatos() Finsi
                Si (opcion = 2) Entonces mostrarResumen() Finsi
        Mientras (opcion <> 3)
Fin
```

Diseño del algoritmo

```
Función agregarDatos
Variables
        distancia, gasolina
        archivo
Inicio
        Escribir "Indique la distancia recorrida: "
        Leer distancia
        Escribir "Indique la gasolina empleada  : "
        Leer gasolina
        archivo.abrir("datos_rendimiento.txt", En Modo de Añadir)
        Si (archivo.falla()) Entonces
                Escribir "Error al almacenar información"
                Terminar
        Finsi
        archivo << distancia << " " << gasolina <<
        archivo.cerrar()
Fin
```

Diseño del algoritmo

```
Función mostrarResumen()
Variables
        archivo
        linea, cuenta
        distancia, gasolina, rendimiento, suma, promedio
        cuenta, flujo
Inicio
        archivo.abrir("datos_rendimiento.txt")
        Si (archivo.falla()) Entonces
                Escribir "No hay datos que mostrar"
                Devolver
        Finsi
        cuenta ← suma ← 0
        Escribir "Distancia", "Gasolina", "Rendimiento"
...
```

Diseño del algoritmo

```
...
        Mientras(obtenerLinea(archivo, linea)) Hacer
                flujo.str(linea)
                flujo >> distancia >> gasolina
                rendimiento  distancia / gasolina;
                Escribir distancia, gasolina, rendimiento
                suma ← suma + rendimiento
                cuenta ← cuenta + 1
        Finmientras
        promedio ← suma / cuenta
        Escribir "Promedio general:", promedio
        archivo.cerrar()
}
```

Elaboración del programa

```
include <sstream>
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

void agregarDatos();
void mostrarResumen();
int main() {
     int opcion;

     do{
          cout << "  MENU PRINCIPAL " << endl;
          cout << " 1. Agregar datos" << endl;
          cout << " 2. Ver resumen  " << endl;
          cout << " 2. Salir        " << endl;
          cout << "Indique una opción: ";
          cin >> opcion;
...
```

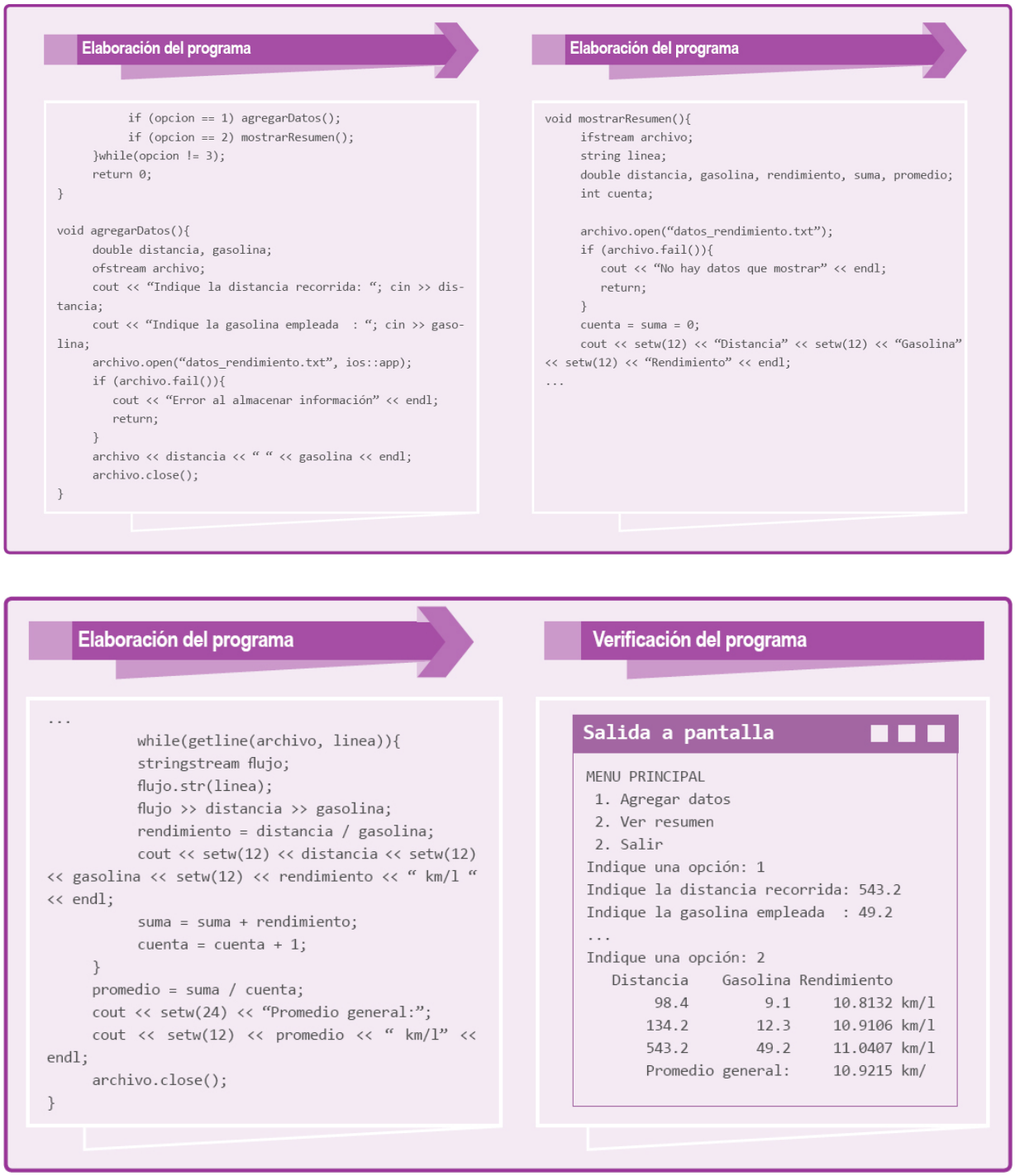


Figura 7. 9. .Solución al problema del registro histórico de rendimiento.

Observe, de la Figura 7.9, las ventajas y seguridad en las operaciones que se llevan a cabo mediante flujos, no sólo de los que representan archivos sino también de los que representan cadenas de caracteres. Del conocimiento de las instrucciones,

facilidades y mecanismos disponibles para la recuperación de la información depende, en gran medida, la eficiencia de los algoritmos que se relacionan con el manejo de archivos.

En la actualidad, los medios de almacenamiento son cada vez más veloces, los discos duros tradicionales han evolucionado significativamente en beneficio de la velocidad de acceso se han eliminado los componentes mecánicos y se han sustituido por componentes electrónicos.

Sin embargo, el uso de los discos duros tradicionales está aún muy extendido y estos representan la pieza más frágil y lenta de los sistemas computacionales actuales. Si a esto se suma que una de las operaciones más lentas es el acceso a disco, se hace evidente la necesidad de un buen diseño en los algoritmos que se enfrentan al requerimiento de recuperar información almacenada de manera persistente.

Con frecuencia las soluciones más eficientes son también las más simples en cuanto a su diseño y concepción; aunque no por ser simples implica que resulten ser las más evidentes.

Se invita al lector a explorar los límites de lo que se ha presentado hasta ahora, a poner a prueba su criterio y a desarrollar su habilidad en el diseño de soluciones eficientes e innovadoras.

7.5 Piense en grande: Centros de datos, océanos de información

En la vida cotidiana con frecuencia se utilizan los servicios de almacenamiento en internet, que proporcionan un espacio para guardar o compartir fotografías, textos u otros elementos de información. Sin embargo, en contadas ocasiones las personas se detienen a pensar en las implicaciones que conlleva este servicio.

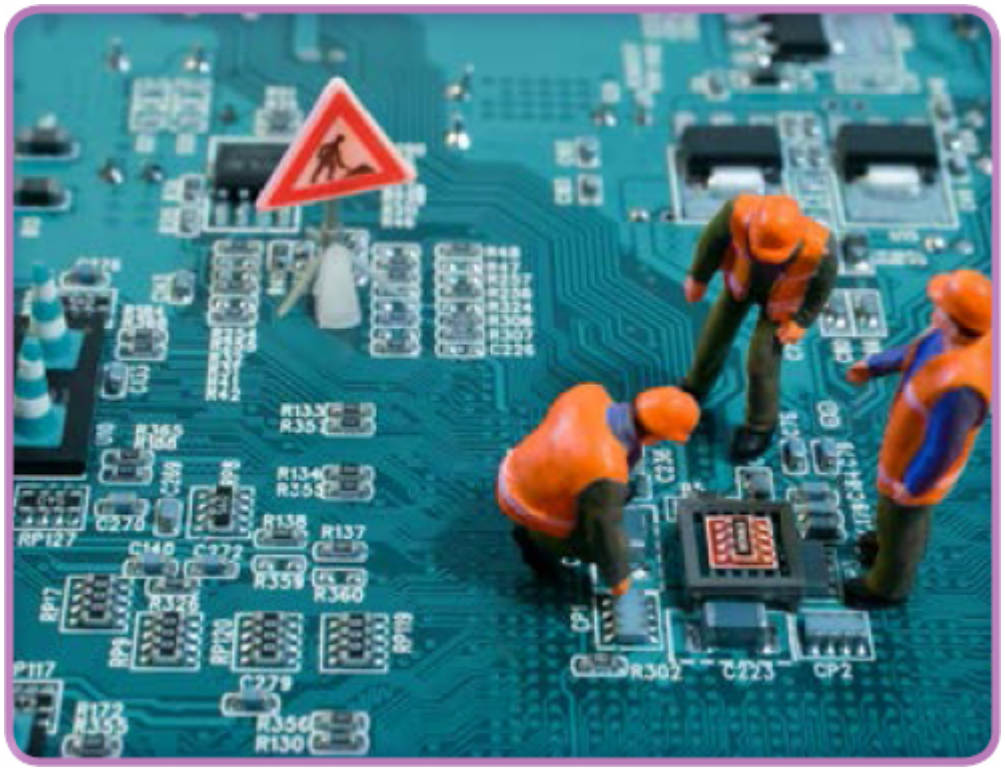
Cada vez es más común que la capacidad de almacenamiento disponible que prestan estos servicios sea tasada en términos de **gigabytes** para cada usuario. Si esto último se multiplica por la cantidad de usuarios y se estima la cantidad de almacenamiento requerido para ese servicio, es posible visualizar que no alcanzaría ni la capacidad de la computadora más poderosa del mundo para almacenar y procesar tal cantidad de información.

¿Cómo hacen, entonces, para funcionar aplicaciones tan famosas y extendidas como las que son utilizadas diariamente? Una buena

parte de la respuesta a esta y otras pregunta relacionadas se encuentra en los *data centers* o **centros de datos**.

Estas enormes estructuras operan y crecen en forma discreta, de manera que más personas puedan seguir gozando del servicio sin importar el crecimiento del mismo. En algunos casos estos centros de datos consumen más energía que ciudades completas; cantidades más que considerables si se piensa en las implicaciones que esto tiene a largo plazo. En el video que se presenta en la barra lateral se observa un poco de lo que hay detrás de estos monstruos de información y de las implicaciones asociadas a su funcionamiento.

Un factor alentador relacionado directamente con el escenario que aparece en el video es que, día a día, las empresas realizan experimentos, diseñan prototipos y respaldan proyectos orientados a hacer más eficientes estos centros de datos. No sólo desde el punto de vista algorítmico, sino desde el punto de vista energético y climático. La escalada de complejidad en estos sistemas ha elevado significativamente el costo para las empresas y para el planeta, por lo que es de esperarse que la tecnología camine en la dirección de reducir en forma drástica el impacto que estos gigantes tienen y podrían tener en el desarrollo tecnológico del planeta.



Actividad de repaso

Problema 1. Archivos

Realizar un programa que presente las tablas de multiplicar del 1 al 10 en pantalla y además que cada vez que se ejecute el programa muestre la leyenda:

```
"Este programa se ha ejecutado n veces"
```

Donde n debe ser sustituido por el número de ocasiones en las que se ha ejecutado el programa. Use un archivo para almacenar información que ayude al programa a determinar el número de veces que ha sido utilizado.

Problema 2. Lista de alumnos

Hacer un programa que almacene nombre y edad de diez alumnos de la clase de programación en un archivo de texto llamado alumnos.txt; realice otro programa que lea la información del archivo alumnos.txt y que calcule el promedio de todas las edades.

Problema 3. Registro de tiempos

Durante el mes pasado, en Austria se realizaron pruebas de calificación con los atletas de la prueba de 100 metros planos; una máquina registraba automáticamente el tiempo del recorrido. Cada uno de los atletas tenía que hacer al menos tres recorridos para ser candidato a participar en los juegos olímpicos.

Una vez terminadas las pruebas mandaron un archivo al comité olímpico internacional con los resultados para ser evaluados, el archivo tiene el siguiente formato:

. . .

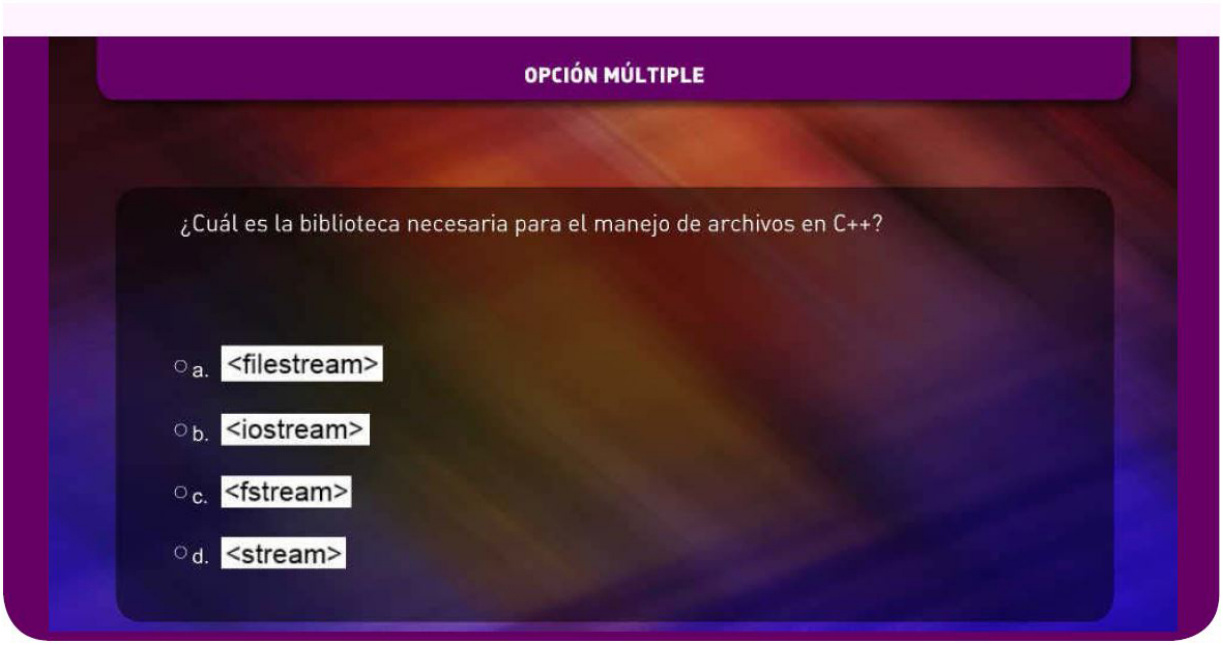
```
Julius Davis, 5, 9.81, 8.91, 9.0, 9.34, 8.39
Andrew Lewis, 3, 9.00, 9.34, 8.99
Bartolome Andersen, 6, 9.12, 9.0, 8.98, 8.99, 9.32, 9.00
. . .
```

En cada línea están los datos de un atleta. El primer dato es el nombre del atleta; el segundo es el número de intentos que hizo; y del tercero en adelante se registran los tiempos que hizo el corredor en cada intento en el recorrido de 100 metros.

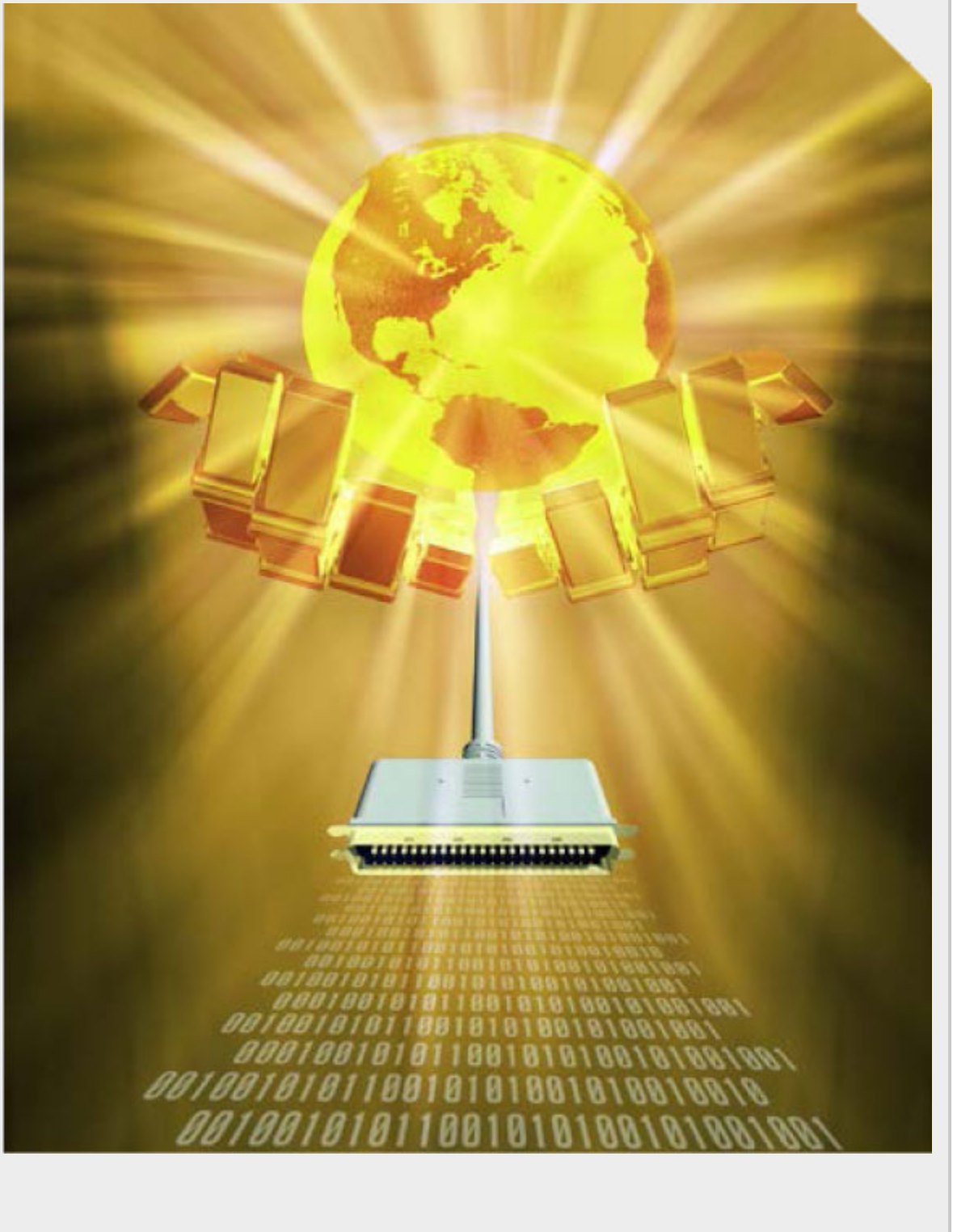
Debes realizar un programa que lea los datos del archivo de entrada y escriba en un archivo de salida el nombre del atleta y el tiempo promedio de los intentos que hizo. El archivo de salida deberá tener el siguiente formato:

```
. . .
Julius Davis, 9.09
Andrew Lewis, 9.11
Bartolome Andersen, 9.06
. . .
```

Ejercicio integrador del capítulo 7



Conclusión del capítulo 7



El acceso oportuno a la información es un factor cada vez más importante de las herramientas tecnológicas y las aplicaciones que se usan actualmente. El manejo eficiente de los grandes volúmenes de información se ha convertido en un requerimiento no nada más en el área computacional, sino también en relación con el consumo de energía y recursos.

Ya sea de manera remota o de forma local, es muy importante que los algoritmos diseñados para la manipulación de la información persistente sean planeados con las restricciones temporales y energéticas actuales. Dado el océano de información en el que el hombre está inmerso el almacenamiento indiscriminado de esa información, si se ignoran las implicaciones posteriores, conduce eventualmente a otros problemas a los que se enfrentan los centros de datos o las mismas personas cuando requieren identificar información valiosa, entre un mar de bytes.

La información y su almacenamiento responsable, cuidadoso y estructurado es un aspecto que hoy en día se considera apenas, muy superficialmente, pero que en los próximos años comenzará a cobrar cada vez mayor relevancia. Ese creciente interés no se debe tanto al costo monetario del almacenamiento en sí, sino a la complejidad inherente en la gestión, organización y búsqueda de información relevante de entre toneladas de bytes almacenados sin distinción.

Glosario general

A

Acumuladores

Variable que incrementa o decrementa su contenido en cantidades variables (acumulador = acumulador + variable). Un acumulador suele utilizarse para almacenar resultados producidos en las iteraciones de un ciclo.

Almacenamiento en la nube

Modelo de almacenamiento en red y en línea donde los datos se guardan en varios servidores virtuales, por lo general organizados por terceros. A los servicios de almacenamiento en la nube se puede acceder a través de una interfaz de programación de aplicaciones (API), o a través de una interfaz de usuario basada en la web.

ANSI

Acrónimo de American National Standards Institute. Una organización voluntaria y sin fines de lucro de grupos de negocios e industria de Estados Unidos formado en 1918 para

el desarrollo de los estándares del comercio y la comunicación. ANSI es el representante americano de la International Standards Organization y ha desarrollado recomendaciones para el uso de lenguajes de programación incluyendo FORTRAN, C y COBOL, así como para diversas tecnologías para redes locales.

Apuntador

Variable que contiene la dirección en memoria de otra variable. Hay apuntadores para cualquier tipo de variable, guardan direcciones y son de mucha utilidad para acceder y manipular datos. También son útiles para pasarle parámetros a las funciones de tal modo que les permiten modificar y regresar valores a la rutina que las llama.

Archivo

Una colección de información completa y con nombre, tal como un programa, un conjunto de datos utilizado por un programa o un documento creado por el usuario. Un archivo es la unidad básica de almacenamiento que habilita a una computadora para distinguir un conjunto de información de otro.

Arreglos multidimensionales

Arreglos que pueden tener 2, 3 ó n dimensiones, aunque sólo se representan visualmente tres dimensiones, con la finalidad de almacenar una lista de elementos del mismo tipo.

B

Biblioteca (de programación)

Es una colección de funciones predeterminadas a las que se hace referencia en un programa. Al compilar el mismo, las funciones se unen al programa principal.

Bit

Abreviatura de dígito binario. La unidad de información más pequeña manejada por una computadora. Un bit expresa un 1 o un 0 en un número binario, o una condición lógica verdadera o falsa, y se representa físicamente por un elemento tal como un voltaje alto o bajo en un punto de un circuito o un punto pequeño de un disco magnetizado de una manera u otra.

Byte

Abreviado como B. Abreviatura de Bynary term (término binario). Un dato que actualmente casi siempre consta de 8 bits. Un byte puede representar un carácter único, tal como una letra, un dígito o una señal de puntuación.

C

Carácter

Es una letra, número o símbolo en programación.

Centro de datos (data centers)

Un conjunto de computadoras de alto rendimiento que provee un servicio. Es una ubicación donde se concentran los recursos necesarios para el procesamiento de la información de una organización.

Ciclo

Un ciclo es una estructura de control utilizada para repetir un conjunto de instrucciones varias veces.

Código ejecutable

Corresponde a las unidades de programas, donde el ordenador puede realizar las instrucciones compiladas mediante el compilador y el enlazador de librerías. Generalmente se confunde con el código objeto, ya que al leer su estructura se comprende como símbolos.

Código fuente

Instrucciones de los programas legibles y que han sido escritos en un lenguaje de alto nivel o ensamblador que no es directamente legible por una computadora. Es necesario compilar el código fuente antes de que una computadora pueda ejecutarlo.

Código objeto

Código generado por un compilador o un ensamblador, que ha sido traducido del código fuente de un programa. Este término se refiere habitualmente a código máquina, que puede ser ejecutado directamente por la unidad central de procesamiento del sistema (CPU), aunque puede tratarse también de código fuente de lenguaje ensamblado, o una variante del código máquina.

Compilador

Programa que convierte un programa fuente, un conjunto de instrucciones en forma de texto, de un lenguaje de alto nivel en código máquina. El código máquina es el lenguaje que el ordenador puede ejecutar directamente.

Computadora de abordo

Conocida también como “módulo de control del tren de potencia” o “unidad de control del motor”, controla muchos de los sistemas del motor de un automóvil, tales como el de combustible y el de encendido. Los sensores y los actuadores detectan el funcionamiento de los componentes específicos (los sensores de oxígeno) y accionan otros componentes (los inyectores de combustible) para mantener un control óptimo del motor.

Contadores

Tipo de variable que incrementa o decrementa su contenido en un valor constante (contador = contador + constante).

Criptografía

Es el estudio de los algoritmos utilizados para cifrar y descifrar información utilizando técnicas matemáticas que hacen posible el intercambio de mensajes de manera que sólo puedan ser leídos por las personas a quienes van dirigidos.

D

Delimitadores

Secuencia de uno o más caracteres para especificar el límite entre regiones separadas. Un ejemplo es el carácter de la coma, que delimita una secuencia de valores. Otro ejemplo son los delimitadores /* y */ que sirven para comentar un texto dentro de un lenguaje de programación.

Dirección de memoria

Identificador único para una ubicación de la memoria, con las cuales una CPU u otros dispositivos pueden almacenar, modificar o recuperar datos. El direccionamiento de la memoria puede considerarse desde dos puntos de vista: físico y lógico. El primero se refiere a los medios electrónicos utilizados en la computadora para acceder a las diversas posiciones de memoria. El segundo, a la forma en que se expresan y guardan las direcciones.

E

Eficiencia temporal o eficiencia espacial

La eficiencia de un algoritmo puede ser cuantificada con las siguientes medidas de complejidad: a. Temporal: tiempo de cómputo necesario para ejecutar algún programa. b. Espacial: memoria que utiliza un programa para su ejecución.

Enlazador

Los ficheros objeto resultantes, junto con las librerías ya existentes y nuevas rutinas de código que se deseen añadir, se

agrupan mediante la operación de enlazado de forma ordenada en un sólo fichero ejecutable.

Estructuras de datos

Forma de almacenar y organizar datos para facilitar el acceso.

Expresión lógica

Expresión que se evalúa a verdadero o falso.

F

Flujo o stream

Es una conexión entre un programa y una fuente o destino de datos. a. Un flujo de entrada maneja los datos que fluyen al programa. b. Un flujo de salida maneja los datos que fluyen del programa.

Función (de programación)

Es un conjunto de instrucciones que llevan a cabo una tarea específica, reciben parámetros formales como datos de entrada y llevan a cabo operaciones con ellos. Cuando un programa hace una llamada a una función, ésta realiza la tarea para la que está diseñada y devuelve el control a la siguiente instrucción del programa desde el que fue llamada.

G

Generador de números pseudoaleatorios

Algoritmo que produce una sucesión de números que es una muy buena aproximación a un conjunto aleatorio de números. Si bien es posible generar sucesiones mediante generadores de números aleatorios por dispositivos mecánicos que son mejores aproximaciones a una sucesión aleatoria, los números pseudoaleatorios son importantes en la práctica para simulaciones y desempeñan un papel central en la criptografía.

Gigabyte

Una unidad de almacenamiento de información cuyo símbolo es el GB que equivale a 109 bytes: a. 1 kilobyte [KB] = 210 bytes = 1024 bytes approx. b. 1 megabyte [MB] = 210 Kbytes = 220 bytes = 1048576 bytes approx. c. 1 gigabyte [GB] = 210 Mbytes = 230 bytes = 1073741824 bytes approx. d. 1 terabyte [TB] = 210 Gbytes = 240 bytes aprox.

GNA (Generador de Números Aleatorios)

Es un programa que crea números de forma aleatoria. Comúnmente se usan algoritmos pseudoaleatorios, es decir, programas que a partir de una semilla, crean una secuencia de números.

I

Palabra

IDE

Acrónimo de Integrated Device Electronics. Tipo de interfaz para unidades de disco, cuya electrónica de control reside en la propia unidad, elimina la necesidad de una tarjeta adaptadora independiente.

Image mining

Conjunto de tecnologías que permiten analizar imágenes y videos del mundo real para obtener información de la misma manera en que lo hacen las personas. Denominado también image analytics, con esta técnica no sólo se analiza la imagen por sus características; realiza además una comprensión integral del entorno en el que está la imagen, lo que permite un análisis más cercano y preciso.

L

Palabra

Lenguaje de programación C++

Extensión del lenguaje de programación C que incluye varias ampliaciones del básico, sobre todo para soportar la programación orientada a objetos. Ha ganado gran aceptación en los últimos años.

Lenguaje de programación

Lenguaje que emplean los programadores para desarrollar aplicaciones específicas. Entre los lenguajes de programación más extendidos están Basic, Cobol, Pascal, C, C++ y Java.

zLínea de comandos

En algunos sistemas operativos de tipo no gráfico, la interfaz permite al usuario introducir, desde el teclado, las instrucciones y mandatos necesarios para la correcta explotación de las tareas que ejecuta el ordenador.

P

Parámetro

La representación abstracta de un dato de entrada en la declaración de una función.

Parsing

Procedimiento que lleva a cabo un analizador sintáctico, convierte el texto de entrada en otras estructuras, más fáciles y útiles para el análisis posterior. Parsing es el proceso de analizar un texto y crear una secuencia de tokens (palabras) para determinar su estructura gramatical con respecto a la gramática formal.

Procesador de texto

Es un software de aplicación en la cual se puede escribir, modificar, eliminar, almacenar texto de manera eficiente y

flexible. Además permite la inserción de imágenes y tablas entre otras funciones.

Programación orientada a objetos (POO)

La POO es un paradigma en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase.

S

Salto de línea

Caracter que indica un movimiento a la siguiente línea de texto. La representación es diferente según la plataforma. En la plataforma Unix, al salto de línea se le denomina nueva línea, en el lenguaje de programación C se escribe \n, en Visual Basic hay un vbNewline para un salto de línea.

Scripting

Secuencias de comandos almacenados como archivos que se pueden ejecutar como programas.

Semántica

Área de la gramática que interpreta el significado de los enunciados generados por el léxico y la sintaxis.

Semilla (del generador de números pseudoaleatorios)

La mayor parte de los generadores de números pseudoaleatorios calcula, o introduce internamente, un valor x0, llamado semilla (número inicial) y, a partir de él, se generan x1, x2,… Siempre que se parta de la misma semilla, se obtendrá la misma secuencia de valores. Por la condición anterior, es evidente que todos los valores generados por este procedimiento son números enteros entre 0 y m-1. El número máximo de cifras distintas que pueden obtenerse con el procedimiento descrito es m, así que llegará un momento en

que el primer número generado se repetirá produciéndose un ciclo.

Servidor

Las computadoras que cuentan con los recursos periféricos reciben el nombre de administrador de la red o server que, auxiliado por el sistema operativo de la red, viene a ser virtualmente el “cerebro” dedicado a administrar los recursos y las comunicaciones entre las demás computadoras. Hay dos tipos de servidores: a. Dedicado. Exclusivamente administra y comparte los recursos de la red. b. No dedicado. Además de administrar los recursos de la red, funciona como estación de trabajo.

Sintaxis

Orden en que deben introducirse los comandos y parámetros de comandos en el indicador del sistema informático. Reglas gramaticales de un lenguaje de programación.

Sistema operativo

Software que controla la ubicación y uso de los recursos de hardware como la memoria, tiempo de la unidad central de procesamiento (CPU), espacio de disco y dispositivos periféricos. El sistema operativo es la base sobre la que se construyen las aplicaciones. Entre los sistemas operativos más populares se encuentran Windows, Mac OS y UNIX.

Stream

Especie de canal a través del cual fluyen los datos. Técnicamente, un stream es el enlace lógico utilizado por el programador en C, C++ para leer o escribir datos desde y hacia los dispositivos estándar conectados a la PC.

Supercomputadora

Computadora de gran tamaño, muy rápida y, con frecuencia, extremadamente cara utilizada para realizar cálculos complejos

o sofisticados.

T

Terabyte

Unidad de medida informática que se representa como “TB” y equivale a: 2 elevado a 40 bytes. En la práctica, un terabyte puede ser tanto 1.000.000.000.000 bytes (10 elevado al 12), como 1.099.511.627.776 bytes (1024 elevado a 4). Equivale a 1.024 veces un gigabyte. 1 terabyte = 1024 gigabytes = 1.048.576 megabytes = 1.073.741.824 kilobytes = 1.099.511.627.776 bytes.

Terminal

Dispositivo que consiste en un adaptador de vídeo, un monitor y un teclado. El adaptador y monitor y, a veces, el teclado normalmente se combinan en una sola unidad. Una terminal realiza poco o nada del procesamiento de la computadora por sí misma; en cambio, se conecta a una computadora con un enlace de comunicaciones sobre un cable. Las terminales se utilizan principalmente en sistemas multiusuario, pues actualmente no se encuentran a menudo sobre computadoras personales de un sólo usuario.

Text mining

Proceso de extracción de información y conocimiento interesante, no trivial desde documentos no estructurados.

U

Palabra

URL

Acrónimo de Uniform Resource Locator. Una dirección para un recurso en Internet. Los URL utilizan los exploradores Web para

localizar recursos de Internet. Un URL especifica el protocolo que se va a utilizar al acceder al recurso (como http: para una página del World Wide Web o ftp: para un sitio FTP – File Transfer Protocol), el nombre del servidor donde reside el recurso (como // www.whitehouse.gov) y, opcionalmente, la ruta al recurso (como un documento de HTML o un archivo del servidor).n

V

Valor de verdad

Valores que pueden asignarse a las expresiones lógicas. Si una proposición es verdadera, se dirá que tiene valor de verdad positivo; si es falsa, negativo. Todo enunciado es verdadero o falso, pero no ambas cosas a la vez.

W

Web mining

Uso de técnicas para descubrir y extraer de forma automática información de los documentos y servicios de la Web. También se puede definir como el descubrimiento y análisis de información relevante que involucra el uso de técnicas y acercamientos basados en la minería de datos (data mining) que se orientan al descubrimiento y extracción automática de información de documentos y servicios de la Web; estos tienen en consideración el comportamiento y las preferencias del usuario.

Referencias

- BCS: The Chartered Institute for IT. (2010). Information Pioneers Contest. Alan Turing by Kate Russell. Recuperado de http://pioneers.bcs.org/pioneer-profiles/alan-turing
- Big think: Experts. (Agosto de 2010). Bjarne Stroustrup Creator of C++: how C++ Combats Global Warming. Recuperado de http://bigthink.com/bjarnestroustrup
- Brokken, F. B. (2011). C++ Annotations. Groningen, The Netherlands: University of Groningen.
- Bulka, D., & Mayhew, D. (2003). Efficient C++: Performance Programming Techniques.: San Diego, California, E.E.U.U.: Pearson Education.
- CppReference. C++ Reference. Recuperado de C++/C++0x/C++11 reference http://en.cppreference.com/w/cpp
- Deitel, P. J. (2009). C++ Cómo programar (6ª ed.). México: Pearson.
- Definición de archivo. (2008). A Dictionary of Computing. Oxford Reference Online. John Daintith & Edmund Wright (Ed.). Oxford University Press, ITESM. Recuperado de http://0-www.oxfordreference.com.millenium.itesm.mx

» Definición de archivo. (2012). In Encyclopedia Britannica. Recuperado el 30 de enero de 2012, de http://0-www.britannica.com.millenium.itesm.mx/EBchecked/topic/206798/file

» Disney Interactive (Pub.). Propaganda Games (Dev.). Welcome to the grid: beyond the code. Tron Evolution. Recuperado de http://www.gametrailers.com/video/beyond-the-code-game-experience/707382

» Downey, A. B. (1999). How to think like a computer scientist C++ (1ª ed.). Boston, Massachusetts, E.E.U.U.: Free Software Foundation.

» Electronics, U. (2011). EE Times. Recuperado de http://www.eetimes.com/design/embedded/4024972/Generating-random-numbers

» Elosua, M., & Plágaro, J. (2007). Diccionario LID Tecnologías de Información y Comunicación. España: LID.

» Google Green. (Mayo de 2011). Google's Hamina Data Center. Recuperado de http://www.youtube.com/watch?v=VChOEvKicQQ

» Google. Google C++ Style Guide. Recuperado el 30 de agosto de 2011, de Google Code: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

» Herb Sutter, A. (2004). C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison Wesley Professional.

» IBM. What is Watson? (s.f.). The Science Behind an Answer. Consultado en http://ibmwatson.com/

» ISO/IEC 14882. ISO/IEC 14882:2011. (3 th ed.). Recuperado de Information technology — Programming languages — C++.

» Josuttis, N. (2004). The C++ Standard Library: A Tutorial and Reference. E.E.U.U.: Pearson Education.

» Kochan, S. (2005). Programming in C. E.E.U.U.: Sams Publishing.

» KXEN's. InfiniteInsight. (Octubre de 2011). What is Predictive Analytics. Recuperado de http://www.youtube.com/watch?v=BjznLJcgSFI

» Little, D. C. (2006). Creating games in C++: a step-by-step guide. Berkeley, California, E.E.U.U.: New Riders.

» Manuales.com. Tecnología. Conecta-T ¿Qué es la criptografía? (s.f). Recuperado de http://www.manuales.com/manual-de/que-es-la-criptografia

» Malik, D. (2007). C++ programming: program design including data structures (3rd ed.). Boston, Massachusetts, E.E.U.U.: Thomson Course Technology.

» Mike Banahan, D. B. (1991). The C Book. E.E.U.U.: Addison Wesley.

» Miles, R. (2001). Introduction to C Programming. Hull, England: The University of Hull.

» Naimipour, R. N. (2011). Foundations of algorithms (4th ed.). Sudbury, Massachusetts: Jones and Bartlett Publishers.

» Nell Dale, C. W. (2005). Programming and problem solving with C++ (4 a ed.). Boston, Massachusetts, E.E.U.U.: Jones and Bartlett Publishers.

» Open Standards. JTC1/SC22/WG21 - The C++ Standards Committee. Recuperado de Open Standards: http://www.open-std.org/jtc1/sc22/wg21/

» Parque de las ciencias. (Prod.). (2011). Ventana a la Ciencia: Genética y Bioinformática. Parque de las Ciencias. Consejo de Educación.Andalucía,Granada. Recuperado de http://www.youtube.com/watch?v=QfeNxJUoIhE

» Plágaro, M. E. (2007). Diccionario LID Tecnologías de Información y Comunicación. España: LID.

» Pozrikidis, C. (2007). Introduction to C++ Programming and Graphics. New York: Springer Science+Business Media, LLC.

» Ramírez, F. (2007). Introducción a la programación: algoritmos y su implementación en VB.NET, C#, Java y C++ (2ª ed.). México: Alfaomega.

» Savitch, W. (2005). Java: an introduction to problem solving & programming. San Diego, California, E.E.U.U.: Pearson Education Inc.

» Savitch, W. (2005). Problem solving with C++: the object of programming (5ª ed.). Boston, Massachusetts, E.E.U.U.: Pearson/Addison Wesley.

» Stroustrup, B. (2000). The C++ Programming Language: Special Edition (3rd ed.). E.E.U.U.: Addison Wesley.

» Suárez, J. M. (2006). Diccionario de comunicaciones (2ª ed.). Medellín, Colombia: Sello Editorial Universidad de Medellín.

» Woodcock, J. E. (2000). Diccionario de Informática e Internet de Microsoft. España: McGraw-Hill Interamericana.

» Wright, E. J. (2008). A Dictionary of Computing. Oxford Reference Online (6th.ed.).Oxford University Press. From Online, Oxford Reference: http://www.oxfordreference.com/

Aviso legal ©

Hernández Alamilla, Sergio Francisco

Fundamentos de programación: un enfoque práctico / Hernández Alamilla, Sergio Francisco

p. 240 cm.

1. Programación de computadoras

LC: QA76.6 Dewey: 001.642

eBook editado, diseñado, publicado y distribuido por el Instituto Tecnológico y de Estudios Superiores de Monterrey.

Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849 | Monterrey, Nuevo León | México.

ISBN en trámite

Edición: Enero 2014