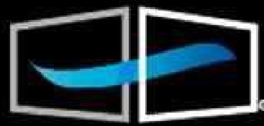


EDITORIAL DIGITAL
TECNOLÓGICO DE MONTERREY

INFORMÁTICA INDUSTRIAL

DR. JOAQUÍN
GUILLÉN RODRÍGUEZ

M. EN C. MAURICIO
HERNÁNDEZ H.

Tabla de contenidos

Acerca de este eBook
Acerca de los autores
Mapa de contenidos
Introducción del eBook
Capítulo 1. Perspectivas generales de consultoría
Introducción
1.1 Fases involucradas en el diseño de un programa
1.2 Técnicas utilizadas en el diseño de algoritmos
1.3 Simbología utilizada en los diagramas de flujo
1.4 Variables y tipos de datos
1.5 Estructuras dentro de los diagramas de flujo
1.5.1 Estructuras secuenciales
1.5.2 Estructuras de selección
1.5.3 Estructuras de repetición
1.5.4 Programación estructurada
1.6 Uso de variables con funciones especializadas
1.7 Diseño básico de diagramas de flujo para solución de problemáticas
1.8 Estructura de un programa en lenguaje C
1.9 Instrucciones y expresiones típicas de lenguajes como C, C++, Java, C#
1.9.1 El operador de asignación
1.9.2 Expresiones aritméticas
1.9.3 Expresiones lógicas o booleanas
1.9.4 Caracteres y cadenas de caracteres (strings)
1.9.4.1 Manejo de caracteres simples
1.9.4.2 Manejo de cadenas de caracteres (strings)
1.9.5 Instrucciones de selección
1.9.5.1 Codificación de if e if-else
1.9.5.2 Codificación de switch
1.9.6 Instrucciones de repetición
1.9.6.1 Codificación de do-while
1.9.6.2 Codificación de while
1.9.6.3 Codificación de for
1.10 Introducción a arreglos
1.10.1 Arreglos unidimensionales (lenguaje C)
1.10.2 Arreglos bidimensionales (lenguaje C)
1.11 Programación orientada a objetos
1.12 Programación gráfica
1.12.1 Introducción a LabVIEW
1.12.2 Alimentando programas con código dentro de LabVIEW
1.13 Programación modular
1.13.1 Implementando la programación modular a través de Labiew (SubVI´s)
Conclusión del capítulo 1
Actividades del capítulo 1
Recursos del capítulo 1
Capítulo 2. Programación gráfica con LabVIEW
Introducción
2.1 LabVIEW como herramienta de programación gráfica
2.2 Estructura while y la ejecución resaltada
2.3 Uso de registros de corrimientos (shift registers)

- 2.4 Temporización en un programa
- 2.5 Túneles
- 2.6 Implementación de estructuras de selección
  - 2.6.1 Estructura Case
  - 2.6.2 Función Select
- 2.7 Uso de valores de controles e indicadores en varias secciones de un programa de LabVIEW (variables locales)
- 2.8 Paralelismo en la programación gráfica
- 2.9 Programación secuencial
- 2.10 Ciclo for y manejo de arreglos

Conclusión del capítulo 2
Actividades del capítulo 2
Recursos del capítulo 2
Capítulo 3. Sistemas de adquisición de datos y arquitectura de las computadoras
- Introducción
- 3.1 Conceptos básicos de adquisición de datos
  - 3.1.1 Equipo de adquisición de datos
  - 3.1.2 Transductores y sensores
  - 3.1.3 Software de adquisición de datos
  - 3.1.4 Señales analógicas y digitales
    - 3.1.4.1 Clasificación de señales
  - 3.1.5 Transmisión de señales analógicas
  - 3.1.6 Transmisión de señales digitales
- 3.2 Adquisición de datos a través de la computadora
  - 3.2.1 Fundamentos de la computadora para la adquisición de datos
  - 3.2.2 La PC para el trabajo en tiempo real
- 3.3 Tarjetas de adquisición de datos
  - 3.3.1 Tarjetas de conversión analógico-digital
  - 3.3.2 Configuración y uso de DAQs con LabVIEW
- 3.4 Puertos de comunicación de la PC
  - 3.4.1 Buses en paralelo
  - 3.4.2 Comunicaciones seriales
  - 3.4.3 Manejo de buses con LabVIEW
- 3.5 Representación y almacenamientos de datos
  - 3.5.1 Manipulación matemática
  - 3.5.2 Gráficas
  - 3.5.3 Archivos

Conclusión del capítulo 3
Actividades del capítulo 3
Recursos del capítulo 3
Capítulo 4. Instrumentación industrial
- Introducción
- 4.1 Automatización y control de procesos industriales
- 4.2 Interfaces hombre-máquina
  - 4.2.1 Importancia e historia de las interfaces hombre-máquina
  - 4.2.2 Características fundamentales de diseño de interfaces
    - 4.2.2.1 Interfaces gráficas para usuarios
    - 4.2.2.2 Interfaces web
  - 4.2.3 Diseño de interfaces gráficas para la adquisición y control de procesos
    - 4.2.3.1 Principios de diseño

4.2.3.2 Desarrollo de menús y esquemas de navegación
4.2.3.3 Organización de ventanas y páginas
4.3 Servicios web en LabVIEW
4.3.1 Definición y características de un servicio web
4.3.2 Implementación de un servicio web a través de LabVIEW
4.4 Introducción a las redes industriales
4.4.1 Conceptos básicos de redes de comunicaciones
4.4.2 Arquitectura de las comunicaciones
4.5 Sistemas de control supervisorio y adquisición de datos
4.5.1 Generalidades de los sistemas SCADA
Conclusión del capítulo 4
Actividades del capítulo 4
Recursos del capítulo 4
Glosario General
Referencias
Aviso Legal ©

Acerca de este eBook

INFORMÁTICA INDUSTRIAL

-

JOAQUÍN GUILLÉN RODRÍGUEZ

MAURICIO HERNÁNDEZ H.

-

D.R.© Instituto Tecnológico y de Estudios Superiores de Monterrey, México. 2013

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

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

Con su nueva 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.

www.ebookstec.com

ebookstec@itesm.mx

Acerca de los autores

JOAQUÍN GUILLÉN RODRÍGUEZ

Profesor del Tecnológico de Monterrey, Campus Tampico. Doctor en Tecnología Avanzada por el CICATA-IPN, maestro en Ciencias con especialidad en Sistemas Electrónicos e Ingeniero en Sistemas Electrónicos del Tecnológico de Monterrey, Campus Monterrey. Cuenta con amplia experiencia académica y de investigación en diversas disciplinas entre las que se incluyen: programación, instrumentación electrónica y control, física y electroquímica.

El Dr. Guillén está certificado como desarrollador asociado en LabVIEW por National Instruments y recibió el premio institucional al mejor software desarrollado en la categoría de posgrado en el Instituto Politécnico Nacional en el año de 2008.

MAURICIO HERNÁNDEZ HERNÁNDEZ

Profesor del Tecnológico de Monterrey, Campus Querétaro. Es ingeniero en Electrónica y Comunicaciones (2004) y maestro en Ciencias en Manufactura con Especialidad en Automatización (2007) por el Tecnológico de Monterrey. Ha trabajado en el departamento de Tecnologías Computacionales y de Electrónica Mecatrónica del Tecnológico de Monterrey desde 2008. En el 2009 participó en la creación de la Cátedra de Investigación en Tecnologías Agrónicas del Tecnológico de Monterrey donde ha desarrollado proyectos en conjunto con la Secretaría de Energía, la secretaría de Agricultura y CONACyT. Es autor de siete artículos de investigación y desarrollo tecnológico en publicaciones internacionales y nacionales así como de un libro en el área de programación gráfica. El Ms. C. es miembro de IEEE y del Project Management Institute.

Actualmente trabaja en el Centro de Ingeniería de General Electric en Querétaro, México donde se

desempeña en el departamento de control de turbinas especializándose en el área administración de energía. Sus principales intereses de investigación incluyen la automatización de procesos agrícolas y las energías renovables.

Mapa de contenidos

Informática industrial

1. Repaso de algoritmos y fundamentos de lenguajes de programación

- Fases involucradas en el diseño de un programa
- Técnicas para el diseño de algoritmos
- Simbología utilizada en los diagramas de flujo
- Variables y tipos de datos
- Estructuras dentro de los diagramas de flujo
- Uso de variables con funciones especializadas
- Diseño de diagramas de flujo para solución de problemáticas
- Estructura de un programa en lenguaje C
- Instrucciones y expresiones típicas de lenguajes como C, C++, Java y C#
- Introducción a arreglos
- Programación orientada a objetos
- Programación gráfica
- Programación modular

2. Programación Gráfica con LabVIEW

- LabVIEW como herramienta de programación gráfica
- Estructura while y la ejecución resaltada
- Uso de registros de corrimientos (shift registers)
- Temporización en un programa
- Túneles
- Implementación de estructuras de selección
- Uso de valores de controles e indicadores en varias secciones de un programa de LabVIEW (variables locales)
- Paralelismo en la programación gráfica
- Programación secuencial
- Ciclo for y manejo de arreglos

3. Sistemas de adquisición de datos y arquitectura de las computadoras

- Conceptos básicos de adquisición de datos
- Adquisición de datos a través de la computadora
- Tarjetas de adquisición de datos
- Puertos de comunicación de la PC
- Representación y almacenamiento de datos

4. Instrumentación industrial

- Automatización y control de procesos industriales
- Interfaces hombre-máquina
- Servicios web en LabVIEW
- Introducción a las redes industriales
- Sistemas de control supervisorio y adquisición de datos

Introducción del eBook

Las tecnologías de comunicaciones e informáticas han influenciado la forma en que vivimos; nuestra sociedad está inmersa en este cambio cultural, el acceso a información y todo tipo de media es ahora más fácil que nunca. Detrás de toda esta revolución se encuentra una gama muy amplia de *software*, *hardware* y medios físicos de comunicación que permiten acceder a tantos recursos.

La llegada de la computadora personal alrededor de la década de los ochenta inició todo este proceso, ahora ha dejado de ser una herramienta individual para convertirse en un equipo que al integrarse a redes de todo tipo provee de casi ilimitadas bondades – y algunas veces también de dependencia enfermiza–. Las laptops, tabletas electrónicas y teléfonos inteligentes, herramientas tan comunes hoy en nuestra vida cotidiana, no son más que el resultado del avance tecnológico de la arquitectura original de la computadora personal.

El incremento exponencial de la capacidad y poder de procesamiento de las computadoras ha permitido también conectarlas a través de sus puertos o de tarjetas especializadas con el mundo físico, esto las ha integrado de manera muy natural como apoyo en los procesos de control industrial. La era digital llegó para quedarse en equipos de automatización y control, como lo son los controladores lógicos programables, interfaces hombre-máquina, sistemas de monitoreo y supervisión (SCADA), etc... Tomando como referencia las redes informáticas, se crearon protocolos y estándares físicos para comunicar y algunas veces permitir acceso remoto o desde internet a equipos industriales. Foundation Fieldbus, Profibus, AS-i, Siemens MPI, etc. son sólo algunos ejemplos exitosos de este tipo de redes.

El propósito de este eBook es fundamentalmente introducir al lector al *software*, *hardware*, equipos, redes y protocolos comúnmente utilizados en la computadora personal y en el área industrial con el fin de que tengan una sólida plataforma para sus cursos avanzados en el área de instrumentación, automatización y control.

Capítulo 1. Perspectivas generales de consultoría

Introducción

En este capítulo se repasarán los conceptos fundamentales de programación, se hará referencia a las estructuras básicas utilizando diagramas de flujo y se detallarán instrucciones (a las cuales también llamaremos sentencias en esta primera parte) típicas de lenguajes de alto nivel, enfocándose la mayor parte del contenido a instrucciones comunes de los lenguajes de alto nivel C, C++, JAVA y C#.

Al ser más de la mitad de este capítulo un repaso referente a estructuras, algoritmos y conceptos básicos de programación, no hay una dirección específica hacia un lenguaje en particular, pero estando la mayoría del material enfocado al análisis estructurado se da preferencia a instrucciones particulares del lenguaje C (estándar ANSI) (Kernighan y Ritchie, 1991; Joyanes y Zahonero, 2005) en ciertas secciones. También cabe mencionar que la sintaxis y semántica de las estructuras algorítmicas del lenguaje C fueron heredadas tanto por C++, Java y C#, por lo que las instrucciones asociadas con estas estructuras son idénticas en todos estos lenguajes.

En particular, el lenguaje C++, al ser un heredero directo de C, respeta todas las características de este lenguaje; por lo tanto, cuando se expresa alguna singularidad o instrucción específica del lenguaje C, implícitamente se aplica a C++. La diferencia principal entre éstos es mencionada al final del contenido temático 1.8.

En la parte final del capítulo se mostrará también, a través del uso del lenguaje gráfico de programación LabVIEW, la manera de integrar instrucciones típicas de un lenguaje de alto nivel así como la implementación del concepto de programación modular.

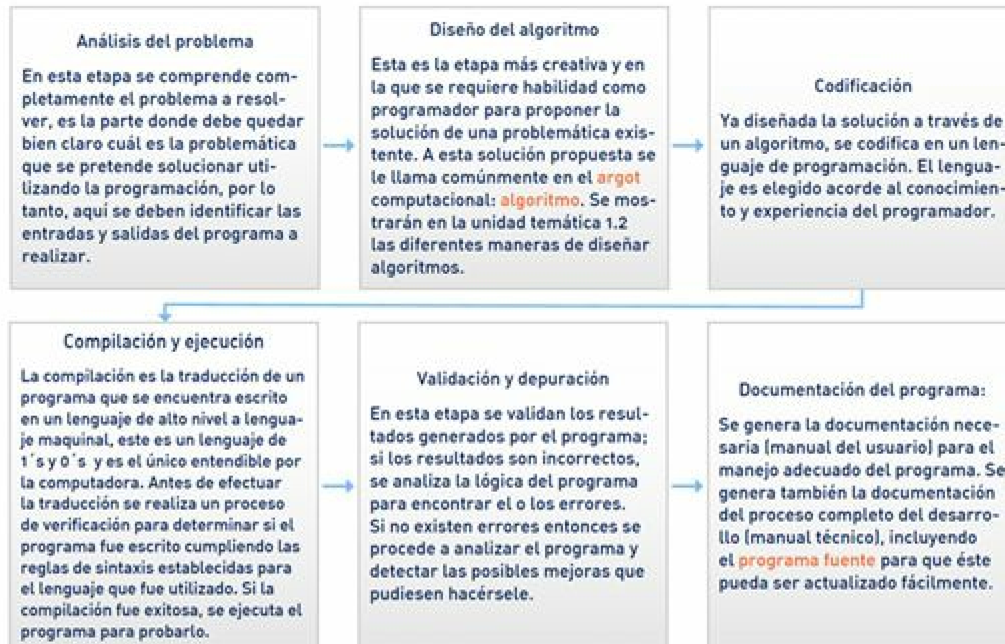
1.1 Fases involucradas en el diseño de un programa

Un programa de computadora es un plan escrito con instrucciones de un lenguaje de programación para resolver un problema. Estas instrucciones le indican a la computadora qué pasos o acciones debe realizar con los datos que se proporcionan para obtener la información deseada.

Básicamente, los programas constituyen el medio de comunicación entre las computadoras y las personas que las utilizan (**usuarios**). Los programas son diseñados por personas capacitadas, empleando la lógica de programación y un lenguaje que facilite la solución del problema.

El desarrollo de un programa es un proceso creativo donde se tiene que pensar de una manera no acostumbrada a hacerlo. Para muchas personas que están aprendiendo a programar esto es un verdadero reto, ya que son obligadas a crear nuevos paradigmas mentales con el fin de solucionar problemas a través de la programación.

A pesar de que el desarrollo de un programa es un proceso altamente creativo, existen varios pasos a efectuar para una buena planeación y desarrollo del mismo; estos pasos facilitan el control del proceso de desarrollo del *software* y se listan a continuación.



Las fases anteriores constituyen la forma más adecuada y recomendable que se debe seguir para desarrollar *software* depurable, escalable y reutilizable.

Revisa la actividad al final del capítulo

1.2 Técnicas utilizadas en el diseño de algoritmos

La parte creativa y, por lo tanto, la más difícil al solucionar un problema a través de un programa de computadora, se encuentran en la etapa del diseño del algoritmo. Comúnmente, en la etapa del desarrollo de un programa, se utilizan dos técnicas: la primera de ellas (y la que se utilizará frecuentemente en esta unidad temática) es la de los **diagramas de flujo**; la segunda técnica es llamada **pseudocódigo** y ambas serán descritas a continuación:

Diagrama de flujo

Un diagrama de flujo es una representación simbólica de la solución de un problema. A través de una serie de símbolos o bloques adecuadamente colocados y conectados con flechas entre sí, se propone una solución para un determinado problema.

Pseudocódigo

Un pseudocódigo es una descripción escrita que indica la forma de solucionar un problema.

La figura No 1.1 nos muestra una representación típica de un diagrama de flujo (1a) y un pseudocódigo (1b).

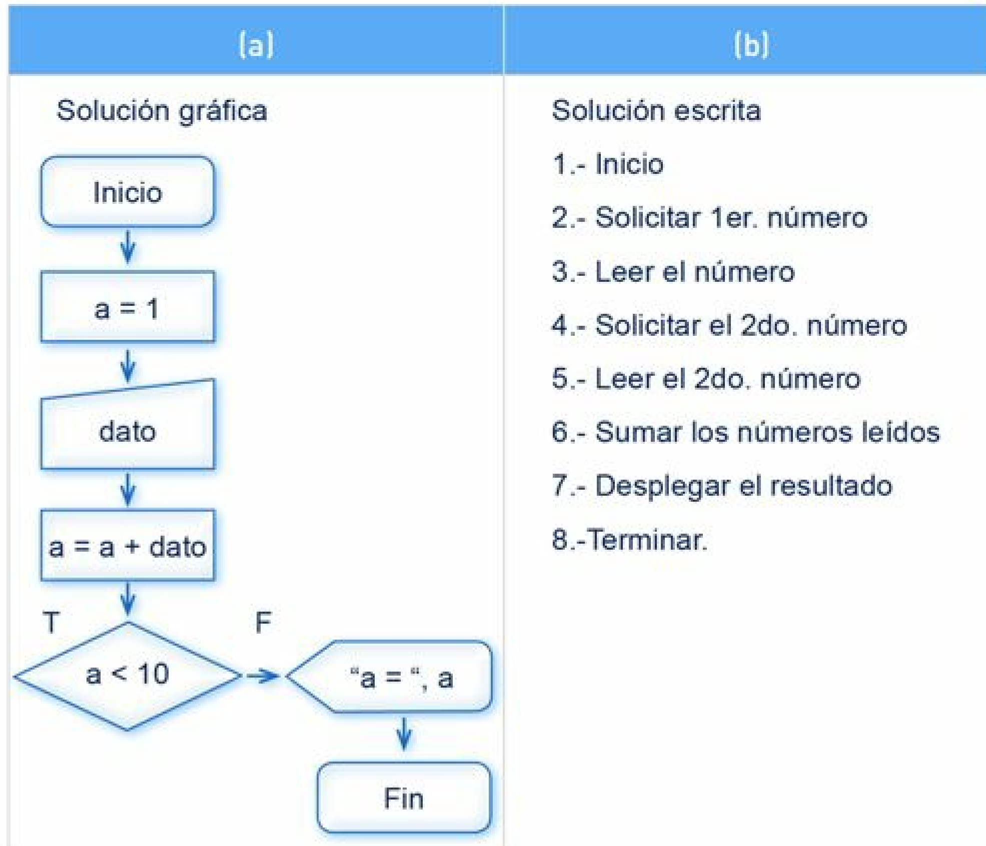
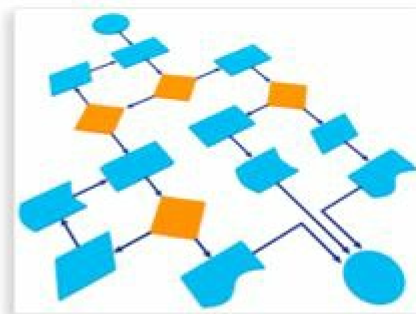


Figura 1.1. (a) Diagrama de flujo (b) Pseudocódigo.

Si pudiéramos establecer una analogía entre un diagrama de flujo y un pseudocódigo con casos de la vida cotidiana para resolver un problema, utilizaríamos el siguiente ejemplo: Un diagrama de flujo se asemeja mucho a las instrucciones de armado de un mueble que viene en forma de un esquema, y donde se va indicando gráficamente cómo se lleva a cabo todo el proceso de armado hasta terminar; Una analogía clásica de un pseudocódigo es una receta de cocina, ya que en ella viene claramente escrito cómo efectuar un platillo determinado.



Tanto la técnica de los diagramas de flujo como la de los pseudocódigos son valiosas para el principiante en el área de la programación, sin embargo se considera que los diagramas de flujo

son más fáciles de captar y entender, por lo que se utilizarán en este capítulo.

Cabe mencionar que muchos de los programadores con experiencia no utilizan ninguna de las técnicas mencionadas para el diseño del algoritmo, esto debido a que han automatizado tanto el proceso de solución de problemas como codifican el programa directamente en la computadora y todo su proceso algorítmico es mental.

Revisa la actividad al final del capítulo

1.3 Simbología utilizada en los diagramas de flujo

Como ya se había mencionado previamente, el conjunto de símbolos que utiliza un diagrama de flujo se une a través de flechas, las cuales indican el sentido que seguirá el control del programa. A continuación se muestran los símbolos básicos que se utilizan en dichos diagramas, así como una descripción del objetivo de cada uno de ellos.

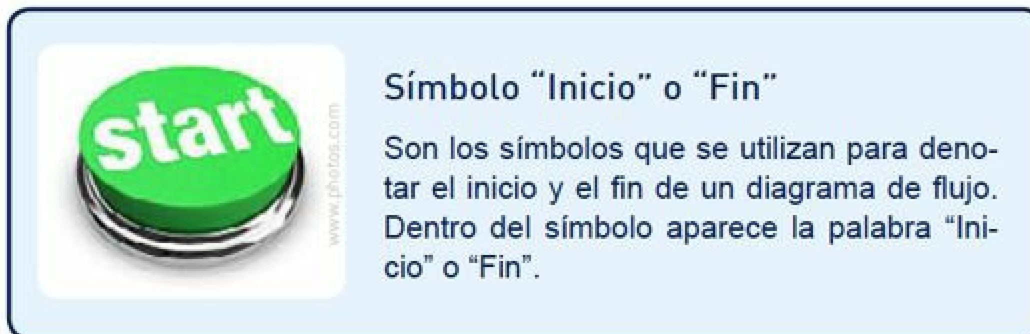
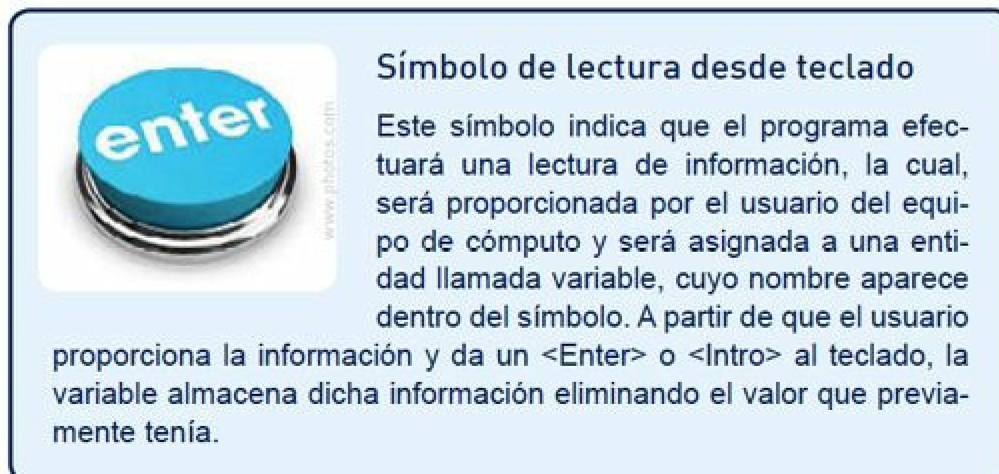


Figura 1.2. Símbolo de Inicio y Fin.



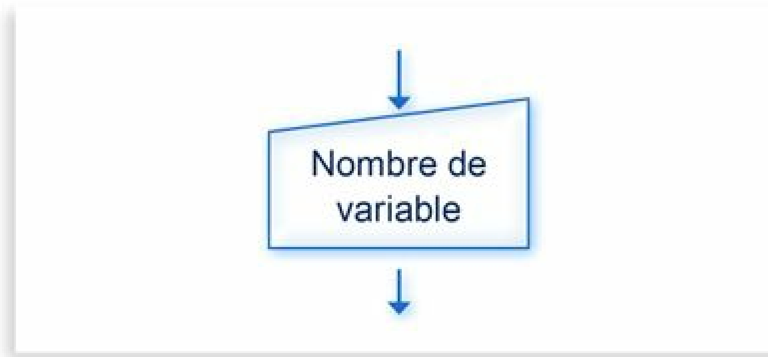


Figura 1.3. Símbolo de lectura desde teclado.

Como nota relevante sobre el símbolo de lectura desde teclado, cabe mencionar que es el único símbolo que detiene el control de un programa. Hasta que el usuario introduce un dato a través del teclado el programa continúa, si el usuario no alimenta ningún dato el flujo del programa se detendrá esperando dicho evento.

Símbolo de proceso

Este símbolo es utilizado comúnmente para asignar a una variable el resultado de operaciones aritméticas, lógicas, relacionales o de otro tipo. Dentro del símbolo aparece una expresión donde se asigna el resultado de una operación a una variable.

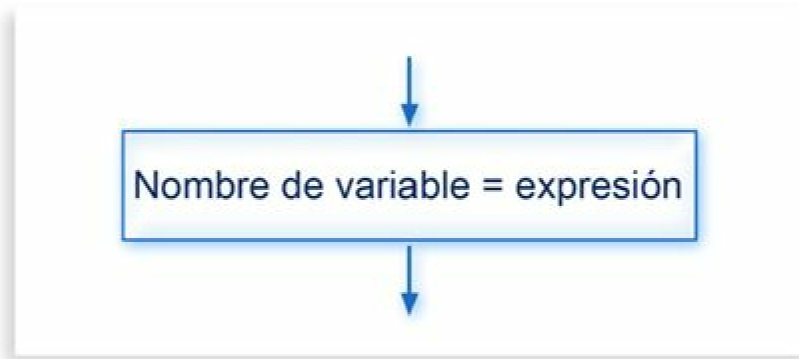


Figura 1.4. Símbolo de proceso.

Símbolo de decisión

Es un símbolo a través del cual se evalúa una expresión lógica o **Booleana** (aquellas que devuelven un valor verdadero o falso) y, de acuerdo a su resultado, el control del programa fluye desde su única entrada hacia una de las dos ramas de salida. Dentro del símbolo aparece dicha expresión y sus dos salidas están marcadas con T (true) y con F (false).

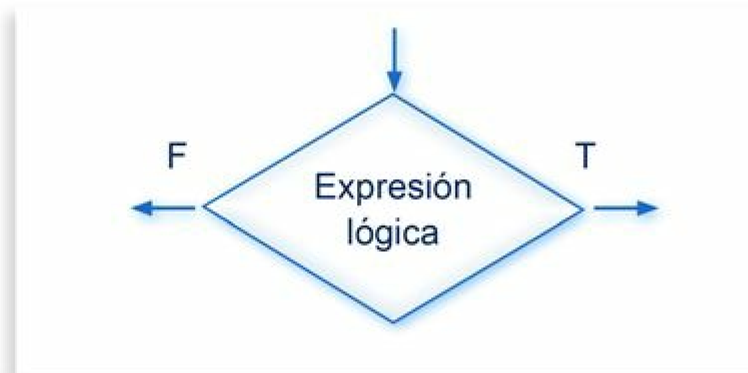


Figura 1.5. Símbolo de decisión.

Símbolo de salida impresa

Este símbolo indica que un conjunto de información de salida será enviada a la impresora. Normalmente la información de salida es separada por comas (,) dentro del símbolo. Si se desea desplegar información de texto, ésta se coloca entre doble comillas ("); si se desea desplegar el valor contenido en una variable se coloca sólo el nombre de esta última.

Figura 1.6. Símbolo de salida impresa.

Símbolo de salida en pantalla

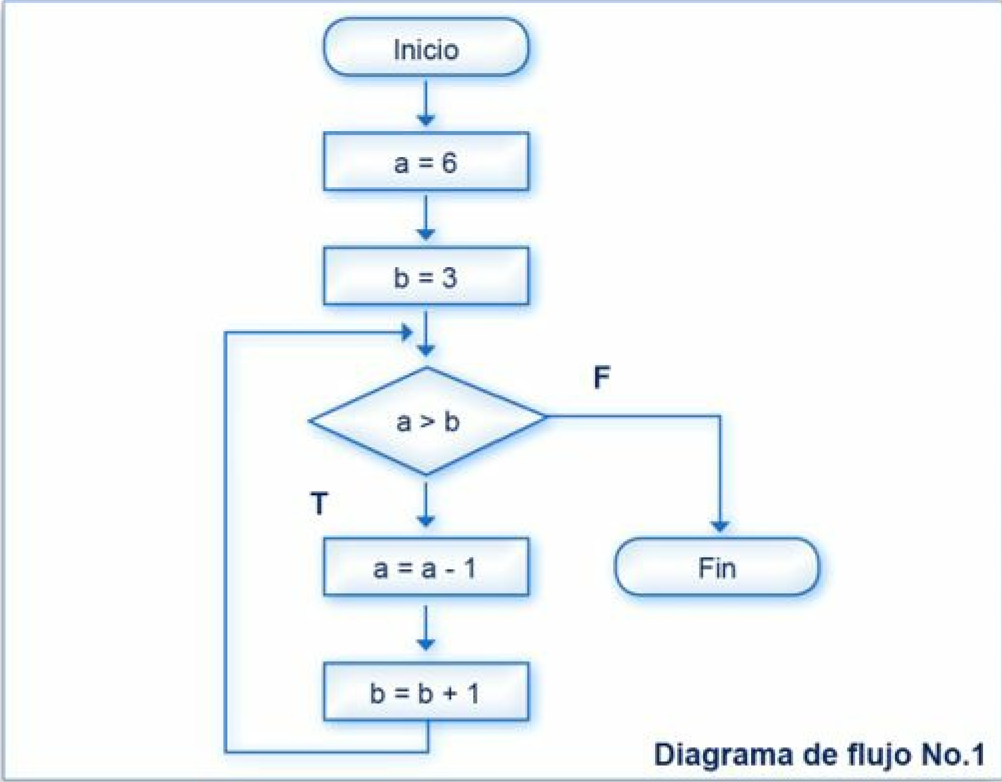
Este símbolo indica que un conjunto de información de salida será desplegada en pantalla. Se maneja la misma nomenclatura que se usa en el símbolo de salida impresa.

Figura 1.7. Símbolo de salida a pantalla.

Figura 1.8. Símbolo de conector.

¿Qué es el control de un programa?

Podemos considerar al control de un programa como una unidad imaginaria que va «ejecutando» cada uno de los símbolos que aparecen en un diagrama de flujo, respetando el sentido indicado por las flechas en el mismo. Esta unidad imaginaria es en realidad la unidad central de proceso (C. P. U. por sus siglas en inglés) del equipo de cómputo, la cual ejecuta las órdenes indicadas por las instrucciones. En la figura 1.9 se muestran 3 animaciones visualizando el comportamiento de la ejecución de un programa dentro de un diagrama de flujo.


Inicio
a = 6
b = 3
a > b
F
T
a = a - 1
Fin
b = b + 1
Diagrama de flujo No.1

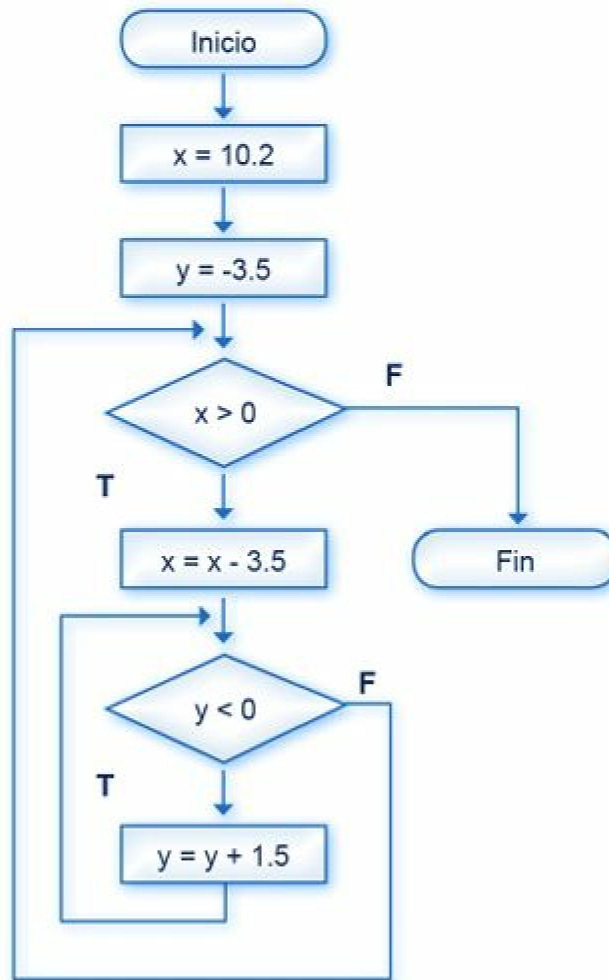


Diagrama de flujo No.2

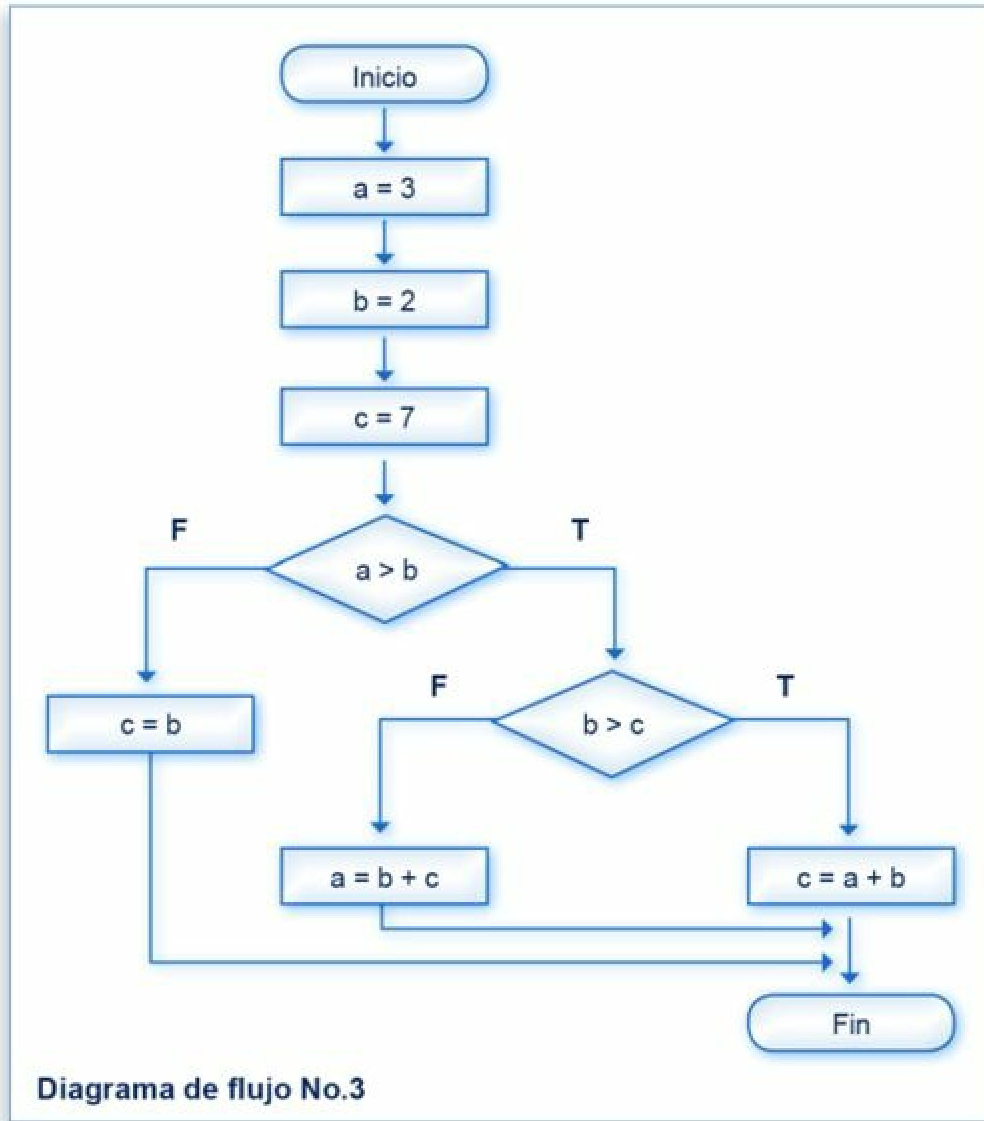


Figura 1.9. Simulación del flujo del control de un programa.

LIGAS DE INTERÉS

Video diagrama de flujo No. 1

Video diagrama de flujo No. 2

Video diagrama de flujo No. 3

Revisa la actividad al final del capítulo

1.4 Variables y tipos de datos

Una **variable** es un espacio reservado en la memoria de la computadora para almacenar información de cierto tipo. Comúnmente se hace referencia a una variable a través de un nombre. Los tipos de variables más comunes en los lenguajes de programación (C, C++, Java, C#) y valores típicos de ellas se muestran en la tabla 1.1.

Tipo de variable	Ejemplo de valores que puede tomar
int	-3, 5, 7, 0, -100
float, double	3.15, -0.00034, 5.6e3, -109.232e-4
String(*)	"Hola a todos", "Saludos", "1234.56", "ht+$3"
char	'H', 'h','L', '$', '.'
boolean(*)	true, false

NOTA:(*) El lenguaje C (versión ANSI)no tiene variables tipo *string*, en realidad este tipo se implementa a través de una estructura de programación llamada arreglo. El tipo Boolean tampoco existe en el lenguaje C, el lenguaje define un valor *true* como cualquier valor diferente de cero y un *false* a un valor igual a cero.

Tabla 1.1. Tipos de datos comunes en lenguajes de programación

Cada lenguaje tiene sus propias reglas o convenciones de la manera correcta de nombrar a una variable; en la mayoría se permite nombrarlas a través de secuencias alfanuméricas de caracteres y muchas veces se permiten caracteres como el guion bajo ("_") u otros. Los caracteres que forman el nombre de una variable no deben estar separados por espacios en blanco, ni se deben nombrar utilizando palabras reservadas por el lenguaje (`int`, `if`, `else`, `while`, `for`, etc.).

Se recomienda siempre que las variables tengan nombres que las identifique fácilmente; por ejemplo, si una variable almacena el impuesto de una transacción, lo más adecuado sería llamar a esa variable `impuesto`, nombres simples o muy cortos no son recomendables (ejemplo: `x`, `ta`, `vd`), ya que en un programa se pierde el sentido de su función.

LIGAS DE INTERÉS

Es un sitio con mucha información respecto a C y C++, se requiere inscripción pero es gratuita. Ejemplos de programas en C y C++, foros de discusión, enlaces a otros sitios, etc., son sólo unos cuantos de los servicios que este sitio ofrece a programadores novatos, intermedios y avanzados.

C Programming and C++ Programming

Sitio oficial de Java, muestra todas las tecnologías alrededor de esta plataforma, tutoriales, descarga de la versión más nueva y su documentación, etc. Indispensable para aquellos que deseen introducirse en el mundo de la programación orientada a objetos.

JAVA

Para usar una variable, ésta se tiene que declarar en un programa y normalmente se permite inicializarla en la misma declaración. Por ejemplo:

```
float iva = 0.16;
```

declara una variable tipo float llamada iva inicializándola en el valor de `0.16`.

La mayoría de los lenguajes hacen diferencia respecto al uso de minúsculas y mayúsculas en el nombre de una variable, esto es, si existe una declaración como la siguiente:

```
int tipo;

char Tipo;
```

Debe quedar claro que para el lenguaje existen dos variables con el mismo nombre y cuya diferencia en este caso es el uso de la mayúscula en la segunda de ellas.

En la mayoría de los lenguajes existen convenciones respecto al nombre de las variables en el estilo de escritura de programas; en el lenguaje C se acostumbra a que las variables sean escritas

completamente en minúsculas y separadas por "_", en caso de que el nombre esté compuesto por más de una palabra; en Java se recomienda que se utilicen mayúsculas en los nombres de variables que estén compuestos por dos o más palabras, al inicio de cada palabra a partir de la segunda. Por ejemplo, el costo total de una transacción en lenguaje C se podría almacenar en una variable llamada `costo_total`; en un programa en Java y para seguir la convención definida por este lenguaje se debería llamar `costoTotal`.

Aunque las convenciones de los lenguajes de programación son sugerencias en el estilo de escritura, el programador serio que desea que sus programas sean legibles, entendibles y escalables a futuro las utiliza y las respeta.

Revisa la actividad al final del capítulo

1.5 Estructuras dentro de los diagramas de flujo

El diseño de un diagrama de flujo es un proceso altamente creativo; siempre será un reto desarrollar la destreza para efectuar diseños de programas bien estructurados y altamente eficientes. Una de las características principales de los diagramas de flujo es que utilizan constantemente estructuras secuenciales, de selección y de repetición; una de las principales tareas de un diseñador de programas es entonces identificar e implementar este tipo de estructuras dentro de un programa.

En este capítulo se define una estructura como uno o más símbolos ordenados de una manera particular dentro de un diagrama de flujo. Se verán a continuación los diferentes tipos de estructuras que se pueden identificar de manera relativamente sencilla en un programa bien estructurado.

1.5.1 Estructuras secuenciales

Como su nombre lo indica, los símbolos dentro de una estructura secuencial están dispuestos en secuencia y se ejecutarán uno a uno de acuerdo al flujo del programa indicado por las flechas. Un esquema de este concepto se muestra en la figura 1.10.

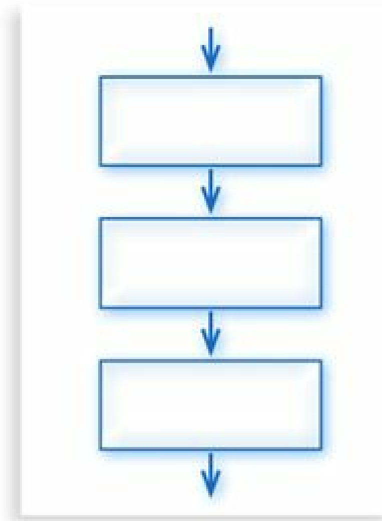


Figura 1.10. Estructura secuencial.

Cabe mencionar que aunque en la figura 1.10 se presentan símbolos de proceso, éstos pueden ser símbolos que representen otra acción diferente (ejemplo: leer un dato, desplegar algo en pantalla, etc…), siempre y cuando sean símbolos que tengan una entrada y una salida.

La **codificación** en un lenguaje de programación de una estructura secuencial sigue el mismo orden que el diagrama, esto es, se codifican las instrucciones correspondientes de cada símbolo dentro del programa.

1.5.2 Estructuras de selección

En programación es muy común que en ciertas condiciones de la problemática a resolver se requiera que se ejecuten ciertas instrucciones y en otras condiciones se ejecute otro grupo de instrucciones, o bien no se ejecute nada, la estructura de selección permite realizar esta tarea.

Las estructuras de selección aparecen dentro de un diagrama de flujo en alguna de las formas mostradas en la figura 1.11.

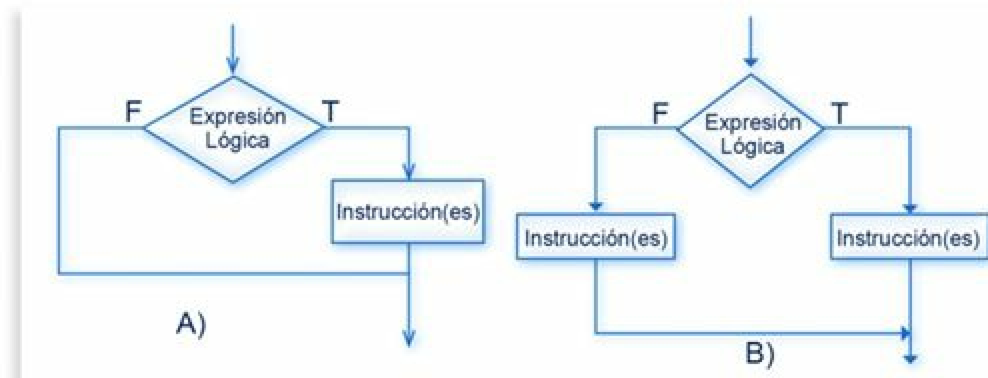


Figura 1.11. Estructuras de selección.

Cuando se desea que el control del programa ejecute un grupo de instrucciones si se cumple una expresión lógica, se plasma la estructura de selección como la figura 1.11(a). Si se desea ejecutar uno de dos grupos de instrucciones de acuerdo al resultado de la expresión lógica la estructura de selección se plasma como lo muestra la figura 1.11(b).

Se verá en la unidad temática 1.9.5.1 que las estructuras de la figura 1.11(a) y (b) se codifican rápidamente en un lenguaje de programación como C, C++, Java y C# a través de instrucciones `if` e `if-else` respectivamente.

1.5.3 Estructuras de repetición

Las estructuras de repetición permiten ejecutar varias veces un mismo grupo de instrucciones. Este tipo de estructuras se identifican por tener lazos que forman gráficamente ciclos dentro de un diagrama de flujo. En todos los casos, el control del programa permanece ejecutando un mismo grupo de instrucciones hasta que una expresión lógica o booleana dentro de un símbolo de decisión es verdadera o falsa, dependiendo del diagrama.

Las estructuras de repetición aparecen dentro de un diagrama de flujo en formas parecidas a la figura 1.12. Nótese que con el sentido de las flechas se forman ciclos en dichas estructuras.

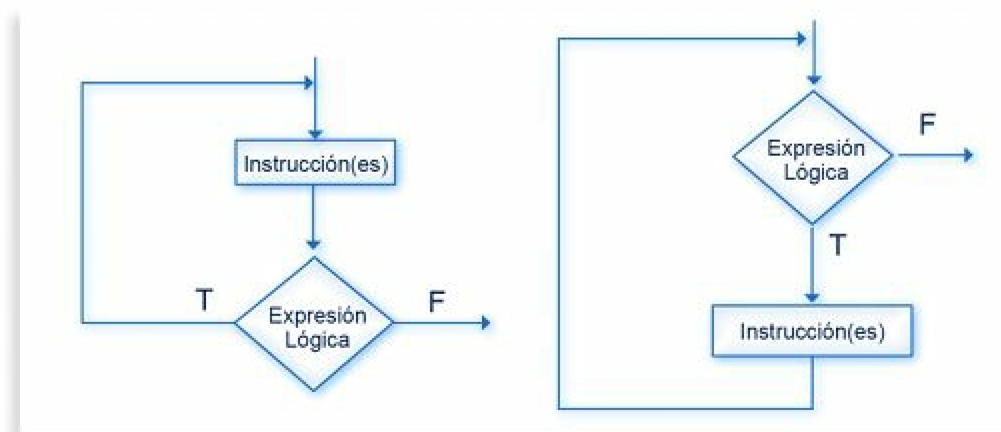


Figura 1.12. Estructuras de repetición.

A pesar de que existen una diversidad más de estructuras de repetición que las que aparecen en la figura 1.12, se verá en el contenido temático 1.9.6 que las estructuras 1.12(A) y 1.12(B) se codifican rápidamente en un lenguaje de programación como C, C++, Java y C# a través de las instrucciones `do-while` y `while`.

1.5.4 Programación estructurada

La programación estructurada es un **paradigma de programación** muy útil en el área de desarrollo de programas; los principios de este paradigma se atribuyen a los primeros trabajos de Edgar W. Dijkstra (Dahl, Edsger y Hoare, 1972). Una de sus premisas establece que en el diseño de un programa, y por lo tanto de un diagrama de flujo, se deben utilizar únicamente las estructuras secuenciales, de selección y de repetición vistas anteriormente. Como consecuencia de lo anterior, el diagrama de flujo, y por ende el programa, se vuelven fácilmente entendibles,

escalables y reutilizables al combinarse adecuadamente estructuras secuenciales, de selección y de repetición. Un punto importante de mencionar es que estas estructuras pueden estar inmersas dentro de otras del mismo tipo o de diferente clase; con esto se puede llegar a cualquier nivel de anidamiento requerido para solucionar una problemática.

Instrucciones existentes en algunos lenguajes como GOTO rompen con los principios de la programación estructurada, por lo que deben de ser evitadas.

La figura 1.13 muestra un diagrama de flujo conteniendo diferentes estructuras identificadas dentro del mismo.

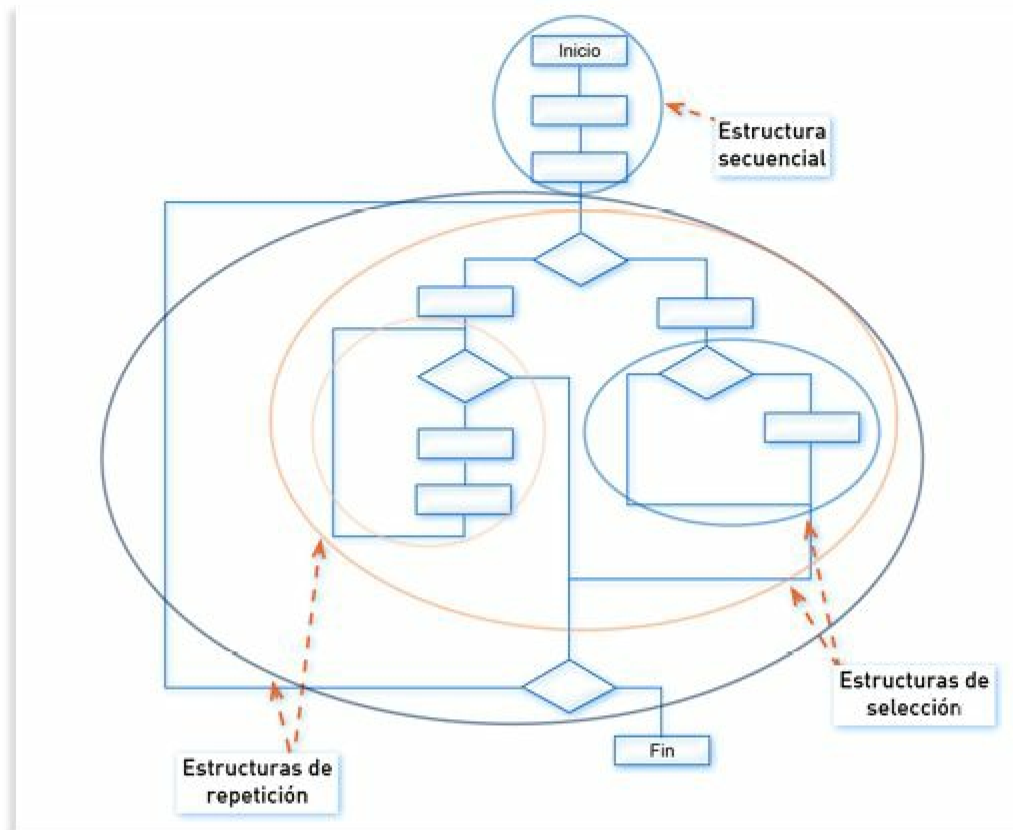


Figura 1.13. Identificación de estructuras dentro de un diagrama de flujo.

Al identificar las estructuras de un diagrama, se puede codificar un programa de manera muy simple. Veremos más adelante que asociando las estructuras con instrucciones específicas del lenguaje de programación la codificación se convierte en una tarea trivial.

Otro aspecto del desarrollo de *software* ligado íntimamente con los principios de la programación estructurada es la programación modular, de ésta se hablará en la unidad temática 1.14.

Revisa la actividad al final del capítulo

1.6 Uso de variables con funciones especializadas

Existen dos clases de variables que son utilizadas muy frecuentemente dentro de los diagramas de flujo y por ende en los programas. La función de estas variables es la misma que las del resto de variables de un programa, esto es, almacenar temporalmente un valor o dato; sin embargo, por la función que realizan se les cataloga con un nombre particular para que su función pueda ser fácilmente identificable en un diagrama de flujo o en un programa.

La primera de las clases es la de las variables contador cuya función consiste en contar eventos que van sucediendo durante la ejecución de un programa; típicamente la instrucción o sentencia asociada a este tipo de operación se encuentra dentro de una estructura cíclica y el valor de la variable se va incrementado cada vez que el control del programa ejecuta la instrucción asociada a este incremento.

Una instrucción como:

```
cont = cont + 1;
```

revela intuitivamente la función de la variable llamada `cont`***, es decir, después de terminar de ejecutar la instrucción, la variable almacenará un valor con una unidad mayor al que tenía antes de ejecutar la instrucción.***

Hay que recordar que en programación comúnmente se ejecuta primero la operación del lado derecho de una asignación y su resultado se almacena en la variable del lado izquierdo, en este caso se suma 1 a la variable `cont` y el resultado se almacena en la misma variable.

La otra clase de variables muy utilizadas en programación son las variables que acumulan información; a este tipo de variables se les llama acumuladores; la función de ellas es la de acumular la información generada en el transcurso del programa. La instrucción asociada a este tipo de operación es similar a:

```
acum = acum + dato ;
```

aquí la variable `acum` se suma a la variable `dato` y su resultado se almacena en `acum`; si se introduce esta instrucción en una estructura cíclica y la variable `dato` se va modificando con diferentes valores dentro del ciclo tendríamos en `acum` el acumulado o la suma de todos los valores que tuvo `dato` dentro del ciclo.

Revisa la actividad al final del capítulo

1.7 Diseño básico de diagramas de flujo para solución de problemáticas

Para un alumno principiante, la solución de un problema a través de un diagrama de flujo es algunas veces complicada, ya que no está acostumbrado a pensar en ciclos, selecciones, etc...

Esta unidad temática está apoyada en cinco videos, la finalidad de ellos es repasar los conceptos básicos y fundamentales de la programación a través del uso de estructuras y su implementación enfocado al lenguaje C. Se pide a los estudiantes revisar los contenidos de los siguientes videos

RECURSOS

Video Programa C1

Video Programa C2

Video Programa C3

Video Programa C4

Video Programa C5

Revisa la actividad al final del capítulo

1.8 Estructura de un programa en lenguaje C

En esta unidad temática se muestra la estructura completa de un programa general en el lenguaje C. Este es un lenguaje que puede adaptarse fácilmente al **paradigma de programación** estructurado, por lo que se describirá de manera más detallada.

Una estructura típica de un programa en lenguaje C la podemos ver en el siguiente **prototipo**

de programa:

```
#include< Archivo de cabecera 1>
#include< Archivo de cabecera 2>  // Declaración de las directivas
#include< Archivo de cabecera 3>  // necesarias a utilizar
. . . . . . . . . .
. . . . . . . . . .
tipo lista_de_variables_globales; // Declaración de las variables
tipo lista_de_variables_globales; // globales a utilizar(OPCIONAL)
tipo lista_de_variables_globales;
. . . . . . . . . .
. . . . . . . . . .

/*Declaración de las funciones auxiliares del programa, esta sección es opcional
  Nota adicional: lo que se encuentra entre corchetes[ ] significa que es
  código opcional, una función puede o no llevar parámetros*/

tipo nombre_de_la_función_auxiliar1([lista_de_parámetros]) {
. . . . . . . . . .
. . . . . . . . . .

}

tipo nombre_de_la_función_auxiliar2([lista_de_parámetros]) {
. . . . . . . . . .
. . . . . . . . . .

}
. . . . . .
. . . . . .

tipo nombre_de_la_función_auxiliar1([lista_de_parámetros]) {
. . . . . . . . . .
. . . . . . . . . .

}
//Declaración de la función main

  int main() {
  tipo lista_de_variables_locales_a_la_función_main;
  tipo lista_de_variables_locales_a_la_función_main;
  tipo lista_de_variables_locales_a_la_función_main;

    instrucción_1;
    instrucción_2;
    instrucción_3;
    . . . . . . .
    instrucción_4;

    return 0;
  } //Fin de la función main y del programa
```

Antes de explicar las diferentes partes del prototipo previo vale la pena comentar que un buen

programador siempre documenta el código escrito en un programa; esa es una tarea que se descuida muy frecuentemente y que causa dolores de cabeza al momento de tratar de escalar un *software* previamente desarrollado. La documentación dentro de un programa se hace a través de comentarios dentro de él; la función principal de estos comentarios es describir las tareas que realiza cada parte de un programa o bien explicar código de programación que pudiera parecer confuso. Los comentarios son completamente ignorados en el proceso de compilación.

En C, C++, Java y C# existen los comentarios de una sola línea y de varias líneas; los primeros se colocan seguidos de "`//`" con lo que cualquier cosa escrita después de estos dos caracteres en la línea es ignorado por el **compilador**. Comentarios de varias líneas inician con "`/*`" en una línea y terminan con "`*/`" en otra línea posterior; cualquier cosa entre estos dos juegos de caracteres es ignorada en el momento de compilar un programa.

> ***El código de un programa en C está compuesto por funciones. La función principal del programa se llama `main`, y es la única indispensable para que un programa pueda ser ejecutado; el resto de las funciones son auxiliares y se emplean o codifican cuando existe una problemática mayor o compleja que resolver.***

Como se muestra en el prototipo, todo programa en C inicia con una serie de directivas **`include`** que indican al compilador integrar archivos(comúnmente con extensión **`.h`**) de definición de diversas funciones necesarias para la ejecución del programa; a estas directivas se les conoce comúnmente como **librerías**. Después de la declaración de estas directivas, se declaran las variables globales del programa precedidas por su tipo (`int`, `char`, `double`, etc.); estas variables se pueden utilizar dentro de todas las funciones del programa.

Siguiendo a la declaración de variables globales se declaran en el prototipo del programa mostrado las funciones auxiliares a utilizar, seguidas por la función principal `main` ; todas estas funciones siguen la misma **sintaxis** de la función `main` dentro de su cuerpo, por lo que no se describirán a detalle.

La función `main` devuelve o regresa un valor de tipo `int`, el cual indica que la función principal del programa regresará un valor entero al sistema operativo; la convención es devolver un cero cuando el programa se ejecutó correctamente y un valor positivo si el programa se interrumpió por algún problema (el valor positivo representa un código de error del programa).

Las funciones auxiliares son de tipo `void` o de algún tipo como `int`, `char`, `float`, `doublé`, etc... Si son del tipo `void` significa que no devuelven ningún valor a la función que llamó a la función auxiliar (esta podría ser `main` u otra función auxiliar), si son de otro tipo deben de regresar o devolver un valor acorde en una instrucción `return`.

Inmediatamente después de la declaración del `encabezado` de la función `main` se colocan entre llaves (“{“ **y** “}”) todas las instrucciones a ejecutar por el programa. Normalmente después de la llave de apertura “{“ se declaran las variables locales a la función `main`; estas variables sólo pueden utilizarse dentro de esta función; en seguida vienen todas las instrucciones del programa y comúnmente antes de la llave final va la sentencia **`return 0,`** la cual indica la finalización de la función `main` y el hecho de que el programa se ejecutó correctamente.

Aunque en el prototipo del programa mostrado las funciones auxiliares se muestran antes que la función `main`, la realidad es que el estilo común es que vayan después; sin embargo esto obliga a que previo a la función `main` deban de declararse lo que se llaman *prototipos de las funciones auxiliares* que no son más que el mismo encabezado de la función terminado en “;”. Por ejemplo, el prototipo de la función auxiliar 1 es:

```
tipo nombre_de_la_función_auxiliar_1([lista_de_parámetros]);
```

El hecho de haber puesto antes de la función `main` a las funciones auxiliares es con el fin de simplificar la explicación.

Finalmente cabe mencionar que para lenguajes como C++, Java y C# su estructura es parecida si se desea programar bajo un esquema de programación estructurada. Sin embargo, es importante comentar que estos lenguajes están diseñados para trabajar bajo un paradigma muy diferente; este paradigma se llama *Programación Orientada a Objetos* y se describirá de manera general en la unidad temática 1.12.

Revisa la actividad al final del capítulo

1.9 Instrucciones y expresiones típicas de lenguajes como C, C++, Java, C#

Cada lenguaje de programación tiene su propio conjunto de instrucciones o sentencias, esta unidad temática se enfoca en las instrucciones, expresiones y tipos de datos más relevantes (y gran parte de las veces comunes) en los lenguajes C, C++, Java y C#.

1.9.1 El operador de asignación

El operador de asignación permite que un valor sea asignado a una variable. La sintaxis de una instrucción conteniendo este operador es la siguiente:

```
variable= expresión;
```

En los lenguajes como Java y C# se obliga a que la expresión sea del mismo tipo que la variable, en C, y por consiguiente en C++, se permiten en ciertos casos asignaciones de tipos de datos diferentes. Cuando un lenguaje permite asignaciones de tipos de datos diferentes se le llama lenguaje de revisión de tipo débil.

En general se sugiere a programadores principiantes asegurarse de que el resultado de la expresión sea del mismo tipo de la variable, ya que de no ser así hay una alta probabilidad de que existan errores en compilación o en ejecución, si se desconoce a profundidad el lenguaje.

En una instrucción de asignación normalmente se ejecuta primero la expresión al lado derecho del operador "=" y el resultado se almacena en la variable ubicada al lado izquierdo del operador; sin embargo, para cada lenguaje existe un orden en la prioridad de ejecución de los operadores (incluyendo el operador "="), por lo que pueden existir en codificaciones de usuarios avanzados expresiones que no se comporten exactamente de esta manera; un ejemplo de una de estas expresiones se muestra en esta misma unidad temática con el uso de operadores postfijos.

Ejemplos del uso de la instrucción de asignación en un programa se muestran en las siguientes líneas de código:

```
int a;
float b,c;
char d;

a=5;
b=4.54326;
c=a/b*2.1-7;
d='h';
```

Existen también dos operadores aritméticos que indican incremento y decremento sobre un operando, estos son "++" y "−−" ; Al aplicar estos operadores a un operando, éste se incrementa o se decrementa en uno; la tabla 1.2 muestra las instrucciones equivalentes utilizando estos operadores.

x++;	equivale a	x=x+1;
x--;	equivale a	x=x-1;
++x;	equivale a	x=x+1;
--x;	equivale a	x=x-1;

Tabla 1.2
Operadores postfijos y prefijos de incremento en 1.

Nótese en la tabla 1.2 que los operadores de incremento y de decremento pueden preceder (prefijos) o seguir a un operando (postfijos). El comportamiento de las operaciones de incremento y decremento dentro de las expresiones varía dependiendo si se utiliza un operador prefijo o postfijo; cuando un operador de incremento o decremento precede a un operando, se realiza la operación de incremento o decremento antes de usar el valor del operando; ahora bien, cuando uno de estos operadores se encuentra después del operando, se utiliza el valor del operando y después hace el incremento o decremento. Por ejemplo, si tenemos en un programa dos líneas de código como las siguientes:

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

después de ejecutarse la segunda instrucción, la variable `y` tendrá `11` debido a que primero se incrementa la variable `x` antes de hacer la asignación. Si tuviéramos el siguiente código:

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

el valor de `y` al finalizar la segunda instrucción será de `10`, ya que primero se usa el valor de la variable `x` y después la incrementa.

Conversión de tipos (casting)

Algunas veces en programación se asigna un valor de tipo diferente a una variable, dependiendo del compilador y/o el lenguaje se pueden generar errores o alertas (warnings) al compilar. Asumamos el siguiente código:

```
float y;
int z;
y=3;
z=y;
```

en muchos compiladores se presenta la siguiente advertencia al compilarlo:

```
warning: converting 'int' from 'float',
possible loss de data
```

La solución para evitarnos estas alertas es utilizar la *conversión explícita* de tipos (también llamada *casting)* de la siguiente manera:

```
float y;
int z;
y=3;
z=(int)y;
```

El propósito del `(int)` antes de la variable `y` es el de convertir explícitamente el valor de la variable `y` a tipo `int` antes de ser usada. La conversión puede ser usada siempre y cuando lo permita el lenguaje y se comprenda perfectamente el uso de la conversión; esto es, no modifique el comportamiento deseado del programa. En el ejemplo del código previo la conversión de un tipo `float` a `int` trunca el valor de tipo `float`, lo cual no trae ninguna consecuencia.

Existen también conversiones implícitas, por ejemplo:

```
int i = 10;
float x = 5.3;
x = x + i;
```

en este código la variable entera `i` es convertida a `float` antes de la suma; esto se hace automáticamente por el compilador.

En general se sugiere buscar en la documentación del lenguaje particular cuáles conversiones de tipo son permitidas, cuáles no, cuáles deben de hacerse explícitamente y cuáles se hacen implícitamente.

1.9.2 Expresiones aritméticas

Una expresión es un conjunto de constantes y/o variables combinadas a través de operadores (una constante única o una variable única se considera también como una expresión). Existen 3 tipos de operadores: los operadores aritméticos, los operadores lógicos o booleanos y los operadores relacionales. Todas las expresiones se evalúan de izquierda a derecha respetando siempre las prioridades de los operadores. Muchas veces dentro de las expresiones existirán paréntesis, en estos casos los paréntesis representan expresiones que deben de ser evaluadas primero, esto se verá en los ejemplos que se muestran más adelante.

Los operadores aritméticos son: `*`, `/`, `+`, `-`, `%`(llamado algunas veces módulo de división entera o residuo). Los operadores `*`, `/`, `+`, `-` actúan normalmente sobre operandos de tipo `float`, `double` ó de tipo `int`.

El operador `%` actúa únicamente sobre operandos tipo int y representa el residuo de la división entera de dos números. Si existe una expresión como `a%b` (`a` y `b` deberían de ser variables tipo `int`) en un programa y las variables `a` y `b` contienen respectivamente 7 y 3, entonces el resultado de esta expresión es ,1 ya que es el residuo de la división.

Cuando el operador de división "`/`" trabaja con operandos enteros, su resultado es entero y es el cociente de la división, es decir, una expresión como a / b (donde `a` y `b` están declaradas como `int`) y contienen respectivamente 5 y 2 es igual a 2, ya que éste es el cociente de la división.

La tabla 1.3 muestra el tipo del resultado al efectuar operaciones aritméticas entre 2 operandos con diversos tipos de operadores aritméticos:

A	OP	B	A OP B
int	+,-,*,/	int	int
int	+, -, *, /	float o double	float o double
float o double	+, -, *, /	int	float o double
float o double	+, -, *, /	float o double	float o double
int	%	int	int

Tabla 1.3. Tipos de los resultados para operaciones aritméticas.

La tabla 1.4 muestra las prioridades típicas de los operadores aritméticos dentro de una expresión; la tabla 1.5 nos muestra diversas expresiones y el valor de su resultado.

++,--	mayor prioridad
-	(unitario)
*, /, %	
+, -	menor prioridad

Tabla 1.4. Prioridades de los operadores aritméticos.

Expresión	Resultado
4 + 3 * 2	10
3 / 4.0 * 10	7.5
5 - 6 * 2 +3	-4
3.5 * 3	10.5
2.5 * 1- 3 * 3.5	-8.0
3 * 10 / 2 % 3 – 2	-2
7 / 4	1
7 % 4	3
12 / 2 - 4 % 5	2
5 * (3 + 2 * 4- 10) / (4 - 2)	2

Tabla 1.5. Ejemplos de expresiones aritméticas y su resultado.

Operaciones abreviadas

Los operadores abreviados se utilizan cuando el usuario empieza a familiarizarse con el lenguaje, esto con el fin de teclear más rápido el código de un programa.

Operador	Instrucción abreviada	Equivalente
+=	m+=n;	m=m+n;
-=	m-=n;	m=m-n;
*=	m*=n;	m=m*n;
/=	m/=n;	m=m/n;
%=	m%=n;	m=m%n;

Tabla 1.6. Operadores abreviados y su uso.

La tabla 1.6 muestra los operadores abreviados más comunes, su uso y el código equivalente del mismo.

1.9.3 Expresiones lógicas o booleanas

Llamaremos aquí expresiones lógicas o booleanas a aquellas expresiones que devuelvan valores `true` (verdadero) o `false` (falso); estas expresiones se forman comúnmente a través del uso de operadores lógicos (and, or, not) y operadores relacionales (<, >, >=, <=, etc.) en una expresión. Incluiremos por lo tanto en esta unidad temática también la explicación y el uso de este tipo particular de operadores.

En particular el lenguaje C maneja las expresiones lógicas de una manera un poco distinta a los demás lenguajes, ya que no posee un tipo lógico predefinido como el tipo `boolean` en Java. Para C una expresión se considera falsa si su valor es 0, en caso contrario la expresión es verdadera. En el resto de los lenguajes se manejan normalmente valores de `true` (verdadero) y `false` (falso) y por lo tanto en ellos queda más claro , simple y fácil de entender el uso de expresiones lógicas dentro de un programa.

Para simplificar la explicación y a la vez abarcar C, C++, Java y C# llamaremos *operandos genéricos* a aquellos operandos que permiten ser identificados por su lenguaje como verdaderos o falsos.

Operadores lógicos

Los operadores lógicos o Booleanos son: "`&&`"(llamado AND), `||`(llamado OR) y "`!`" (llamado NOT). Estos operadores actúan comúnmente sobre operandos genéricos. Los operandos && y || actúan sobre 2 operandos mientras que ! actúa sobre uno solo. La tabla 1.7 muestra las tablas de verdad de estos operadores.

A	B	A && B	A || B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

A	!A
false	true
true	false

Tabla 1.7.
Tablas de verdad para los operadores lógicos.

En el caso del lenguaje C, el cual es un lenguaje con revisión de tipo débil, se devuelve un uno o un cero cuando se hace una operación lógica. Por ejemplo, para el siguiente código:

```
int a,b,c;
a = 3;
b = 0;
c = a && b;
```

El valor final de la variable `c` sería 0 ya que la variable `a` contiene 3 (es verdadero para el lenguaje C); la variable `b` contiene 0 (es falso para el lenguaje C), por lo que de acuerdo a la tabla 1.7 "`a && b`" será falso (0 en el lenguaje C). Cabe hacer notar que la última instrucción del código mostrado marcaría un error en compilación en otros lenguajes con revisión de tipo fuerte.

Obsérvese de la tabla 1.7 que el operador "&&" (AND) hace la expresión verdadera (true) sólo si ambos operandos son verdaderos (true), en caso contrario el resultado de la expresión es false. Para el operando "||" (OR) el resultado de la expresión es verdadero (true) si al menos uno de los operandos es verdadero (true), en caso contrario el resultado de la expresión es falsa (false). El operador! (NOT) "niega" o complementa el valor de un operando, esto es, si el operando es verdadero (true) la expresión "!operando" es falsa(false) y si el operando es falso(false) la expresión "!operando" es verdadera(true).

Una expresión lógica se puede asignar a una variable tipo boolean en la mayoría de los lenguajes, sin embargo, como el lenguaje C carece de este tipo de dato, el resultado se puede almacenar en un tipo de dato int como se mostró en un código previo.

Las prioridades de los operadores lógicos dentro de una expresión se muestran en la tabla 1.8

!(NOT)	mayor prioridad
&&(AND)	prioridad intermedia
||(OR)	menor prioridad

Tabla 1.8. Prioridades de los operadores lógicos o booleanos.

Ejemplos de expresiones usando operadores lógicos en el lenguaje C se muestran en la tabla 1.9.

Expresión	Resultado
a || b && c	1
a && b && c	0
a && (b || d) && c	0
a || c	1
! b && ! (a || d)	0
! c	1

Nota: Asuma que a, b, c y d son variables de tipo int y que tienen valor de 3, 0, 0 y 1 respectivamente al efectuarse las expresiones.

Tabla 1.9. Ejemplo de expresiones usando operadores lógicos en el lenguaje C.

Operaciones relacionales

Los operadores relacionales son: “>” (mayor), “<” (menor), “>=”(mayor o igual), “<=”(menor o igual), “`!=`”(diferente) ,”==”(igual). Los operadores relacionales actúan sobre operandos de cualquier tipo y permiten establecer una comparación del valor de dichos operandos. El resultado de una operación que contenga cualquiera de los operandos relacionales es booleana, es decir, es verdadera (`true`) si el resultado de la comparación es verdadera, y falsa (`false`) si no lo es, o bien 1 y 0 en el caso particular del lenguaje C.

En la tabla 1.10 se muestran ejemplos de expresiones involucrando operadores relacionales.

Expresión	Resultado
a >= b	false(0 para C)
b < c	true(1 para C)
c = = 5	false(0 para C)
e != d	true(1 para C)
4 > 3	true(1 para C)
7.5 <= 2.3	false(0 para C)

Nota: Asuma que a, b y c son variables de tipo int con valores -4, 5 y 7 respectivamente, d y e son variables de tipo float con valores -0.5 y 9.2 respectivamente)

Tabla 1.10. Ejemplo de expresiones usando operadores relacionales.

Algunas veces será necesario realizar expresiones que contengan simultáneamente los 3 tipos de operadores (aritméticos, lógicos o booleanos y relacionales); en este caso la tabla de prioridades de todos ellos se muestra en la tabla 1.11 (se incluye también el operador de asignación):

<table>
<tr><th>Operador</th><th>Prioridad</th><th>Dirección de la asociación</th></tr>
<tr><td>++,--, !,+ (unitario),- (unitario)</td><td>mayor prioridad</td><td>derecha</td></tr>
<tr><td>*, /, %</td><td></td><td>izquierda</td></tr>
<tr><td>+, -</td><td></td><td>izquierda</td></tr>
<tr><td><, <=, >, >=, =|</td><td></td><td>izquierda</td></tr>
<tr><td>= =, !=</td><td></td><td>izquierda</td></tr>
<tr><td>&&</td><td></td><td>izquierda</td></tr>
<tr><td>||</td><td></td><td>izquierda</td></tr>
<tr><td>=</td><td>menor prioridad</td><td>Derecha</td></tr>
</table>

Nota: Es importante confirmar la información de esta tabla de acuerdo al lenguaje con el que se esté trabajando

Tabla 1.11. Prioridades típicas de operadores dentro de una expresión.

En la tabla 1.12 se muestran ejemplos de expresiones involucrando diversos tipos de operadores junto con sus resultados.

Expresión	Resultado
(a>b) && (c<=d)	false(0 para C)
a+b*c < d-a	true(1 para C)
! ((a > b+c) || (d <4) && (e>a))	true(1 para C)

Nota: Asuma que a,b,c y d son variables tipo `int` y que tienen valores -2, 6, -4, 8 respectivamente

Tabla 1.12. Expresiones involucrando diversos tipos de operadores.

1.9.4 Caracteres y cadenas de caracteres (*strings*)

El manejo de caracteres simples y únicos a través de variables tipo `char` es casi idéntico dentro de los lenguajes C, C++, Java y C#; los grupos o cadenas de caracteres también llamado **strings** en programación difieren en su operación acorde al lenguaje utilizado. En esta unidad temática se muestra el manejo de caracteres y de *strings* enfocándose en particular al lenguaje C.

1.9.4.1 Manejo de caracteres simples

Un carácter simple puede ser almacenado dentro de una variable tipo `char`; un ejemplo de la declaración e inicialización de una variable de este tipo es la siguiente línea de código:

```
char respuesta = 'S' ;
```

En la línea de código previa se declara la variable llamada respuesta y se inicializa con el carácter 'S' mayúscula. Los caracteres se almacenan internamente a través de un código que ocupa un byte para cada carácter en el lenguaje C, el código almacenado es llamado ASCII el cual es un código de 7 bits, por lo que codifica únicamente 128 caracteres; de estos, algunos son caracteres de control y el resto otros son imprimibles o desplegables, el apéndice 1.1 muestra una tabla de todos los caracteres ASCII con sus códigos correspondientes en decimal, octal y hexadecimal.

Cuando existe una comparación entre 2 variables tipo `char`, lo que realmente se compara son los códigos de los caracteres que contiene cada una de las variables. Al hacer una revisión de la información del apéndice 1.1 se puede deducir que una variable tipo `char` que contenga un digito ('0'..''9') como carácter almacenado será menor en una comparación respecto a una que tenga un carácter alfabético en mayúscula ('A'..''Z'), y ésta última variable será menor que una que contenga un carácter alfabético en minúscula ('a'..'z').

RECURSOS

Apéndice 1.1

C es un lenguaje de revisión de tipo débil, esto es, permite en muchas ocasiones la transferencia de datos directamente entre variables de tipo diferente. Debido a esto, las variables `char` e `int` pueden ser intercambiadas en su uso de manera simple, esto no genera problemas siempre y cuando la variable tipo `int` tenga valores entre 0 y 127. Un código como el siguiente ejemplifica este aspecto del lenguaje:

```
int i=5
char ch='X';
ch = ch + 1;    //incrementa el código de ch en 1, ahora ch
                //almacena el carácter 'Y'
i = ch;         //se almacena un 89 en la variable i
printf("%d", ch);  //se despliega el código de ch (89)
printf("%c",i);    //se despliega el carácter 'Y'
```

Los lenguajes modernos como Java son muy diferentes en este aspecto; las instrucciones después de la declaración de las variables `i` y `ch` serían incongruentes en este lenguaje y generarían errores en la etapa de compilación.

En el lenguaje C existen muchas funciones auxiliares que manipulan y/o prueban caracteres, la tabla 1.13 muestra algunas de estas funciones y su significado semántico.

Nombre de la Función	Significado dentro del lenguaje C
`isalnum(ch)*`	¿Es un número?
`isalpha(ch)*`	¿Es un carácter alfanumérico?
`iscntrl(ch)*`	¿Es carácter de control?
`isdigit(ch)*`	¿Es dígito decimal?
`isgraph(ch)*`	¿Es carácter de impresión excepto espacio?
`islower(ch)*`	¿Es letra minúscula?
`isprint(ch)*`	¿Es carácter de impresión excepto espacio, letra o dígito?
`isspace(ch)*`	¿Es espacio, avance de línea, nueva línea, retorno de carro o tabulador?
`isupper(ch)*`	¿Es letra mayúscula?
`isxdigit(ch)*`	¿Es dígito hexadecimal?
`tolower(ch)`	Función que devuelve o regresa el código de ch en minúscula
`toupper(ch)`	Función que devuelve o regresa el código de ch en mayúscula

Nota: ch es una variable char o int (con un valor entero entre 0 y 127), (*) las funciones devuelven un valor diferente de 0 cuando son verdaderas o 0 cuando son falsas. Las últimas dos funciones devuelven el código en minúscula/mayúscula del carácter o código almacenado en ch.

Tabla 1.13. Funciones asociadas con el manejo de char´s en el lenguaje C.

Todas las funciones de la tabla 1.13 se encuentran en el archivo de encabezado (librería) de C llamado `ctype.h`, por lo que debe especificarse este archivo a través de la directiva `include` en el inicio del programa a desarrollar.

El código ASCII está siendo sustituido recientemente por **UNICODE**, ya que éste es un código de 16 bits (2 bytes) y puede integrar fácilmente los caracteres de todos los idiomas del planeta. Para mayor compatibilidad se decidió que los primeros 128 caracteres correspondieran a los mismos caracteres codificados por ASCII.

1.9.4.2 Manejo de cadenas de caracteres (strings)

Una cadena de caracteres-llamada también *string*- dentro del lenguaje C no es más que una variable que almacena un conjunto de caracteres agrupados en una estructura de programación llamada **arreglo**. Esta estructura se verá formalmente en el contenido temático 1.11. Para declarar un *string* de un tamaño determinado, se utiliza la sintaxis mostrada en el siguiente ejemplo:

```
char str[30];
```

En esta declaración la variable llamada str es un *string* declarado de tamaño 30, es decir, puede almacenar como máximo 30 caracteres, sin embargo, cuando se desea almacenar y manejar un mensaje como una unidad, más que caracteres individuales independientes entre ellos se reserva un espacio para colocar al final del mensaje un carácter especial llamado *el* carácter nulo *'\0'*, por lo que en este caso se tiene capacidad para 29 caracteres.

Una declaración como:

```
char ch;
```

declara que la variable `ch` puede almacenar únicamente un caracter simple a diferencia de un *string*, el cual puede almacenar múltiples caracteres.

Una variable *string* puede utilizarse directamente con instrucciones printf y scanf para cargarse de información o para desplegar su contenido en pantalla; las últimas dos instrucciones de las 3 líneas de código siguiente ejemplifican el uso de estas instrucciones en el lenguaje C.

```
printf("dame un mensaje. . . . .");
fgets(str,sizeof(str)-1,stdin);
```

Por *default* la instrucción `scanf` lee el texto hasta un **delimitador**, y como el espacio en blanco se considera delimitador, trunca el *string* si se da un mensaje con espacios en blanco. Si se desea almacenar un mensaje en la variable *string* llamada `str` que contenga espacios en blanco, se debe de usar la instrucción `fgets(str,sizeof(str)-1,stdin)` en vez de `scanf`; las siguientes líneas de código muestran cómo hacerlo:

```
printf("dame un mensaje. . . . .");
fgets(str,sizeof(str)-1,stdin);
```

La función `sizeof` dentro de la segunda línea de código previo obtiene el número de bytes reservados para el *string* en la memoria, en nuestro caso 30, sin embargo se le resta uno para indicar que el máximo tamaño leído del *string* será de 29 caracteres recordando que el sistema al ejecutar el código agregara el carácter nulo "\0"). En resumen, la instrucción `fgets` lee de teclado – definido como la entrada estándar (`stdin`)- cuando mucho 29 caracteres de texto.

Acceso de elementos individuales de un *string*

Para accesar un carácter específico del *string* se utiliza un índice que tiene un valor entero que puede ir desde 0 hasta la longitud del *string* –1, por eso la expresión `str[0]` (índice 0) especifica el primer caracter del *string*, `str[1]` (índice 1) el segundo carácter del *string*, `str[2]` (índice 2) el tercer carácter y así sucesivamente. Si el usuario da un mensaje como "Hola México" a la ejecución del código previamente mencionado la información se almacena como lo muestra la figura 1.14.

str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]	str[7]	str[8]	str[9]	str[10]	str[11]
'H'	'o'	'l'	'a'	' '	'M'	'e'	'x'	'i'	'c'	'o'	'\0'

Figura 1.14. Almacenamiento de caracteres individuales en un String.

El carácter nulo ('\0') se añade al *string* cuando el usuario da Intro o Enter. El resto de los

elementos (de str[12] a str[29]) tienen información aleatoria o información previa dejada por el programa en caso de que se hubiese ejecutado anteriormente otra instrucción fgets.

El tamaño o número de caracteres que contiene almacenado un *string* puede calcularse con el uso de la función strlen(de STRing LENgth); ésta regresa o devuelve el número de caracteres que tiene el mensaje almacenado en el *string* . Para utilizarla debemos de incluir en una directiva include al inicio de nuestro programa la librería string.h. Para ejemplificar el uso de la función strlen revisa las siguientes líneas de un programa, esta rutina es equivalente a utilizar la instrucción printf(”%s‟,str).

```
n=strlen(str);  // las variables i y n deben de
                // estar declaradas como int
i=0;
while (i<n) {
      ch=str[i]; //se almacena el carácter de la posición i
                 //en la variable char llamada ch

         printf("%c",ch) ; //se manda desplegar a pantalla
                           // la variable ch
      i++;       // se incrementa la variable i
}
//el código previo es equivalente a usar
// una instrucción printf(”%s‟,str);
//sólo que en vez de desplegarse en una sola instrucción
//se desplegó carácter por carácter a través de un ciclo
//while
```

Una de las cosas que nos indica el ejemplo previo es que se pueden manipular los caracteres individuales de un mensaje a través de la notación:

```
variable_string[indice]
```

donde `variable_string` es el nombre de un *string* (arreglo de chars) e `índice` es una constante numérica o variable entera, conteniendo un valor entre 0 y la longitud del mensaje almacenado en el *string* - 1.

Por ejemplo, si deseamos realizar un programa que determine cuántas veces aparece un carácter específico dentro de un mensaje, podríamos efectuarlo con el código que precede a este párrafo, este código se apoya con una función auxiliar llamada cuenta; esta función es de tipo `int`, es decir, regresa un valor entero a la función que la llamó. Normalmente, cuando una función regresa un valor, éste es almacenado en una variable para ser utilizado posteriormente; la función cuenta recibe dos **parámetros**, el primero es el carácter a buscar y el segundo es el *string* donde será buscado el carácter. La función "regresará" al programa el número de caracteres encontrados en el *string*.

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

/*Se decñara ima función llamada cuenta que recibe un caracter
  y un string como entrada y devuelve la cantidad de veces
  que e caracter se encuentra dentro del string*/

 int cuenta(char ch, char str[]){//Nótese como se define un parámetro
                                 //que es un arreglo
 int i,contador;
   contador=0;
   for(i=0;i<strlen(str);i++)
      if(str[i]==ch) contador++;//si el carácter analizado dentro del
                                //string coincide con el
                                //carácter buscado
                                //se incrementa la variable contador

   return contador;   //nótese que el return no va sólo como en
                      //una función tipo void sino que
                      //va acompañado del valor que
                      //"regresa" la función, en este caso
                      //el valor de la  variable contador
 }//fin de la función llamada cuenta


 int main(){
   char carácter;
   char mensaje[50];
   int numero;

   printf("dame un mensaje. . . . .");
   fgets(mensaje,sizeof(mensaje)-1,stdin);
   printf("carácter a buscar dentro del mensaje....");
   scanf("%c",&carácter);
   numero=cuenta(carácter,mensaje); //recuerda cómo se efectúa
                                    //la llamada a la función
                                    // cuenta, como ésta regresa un
                                    // valor se puede almacenar
                                    //éste en una variable.

   printf("se encontraron en el mensaje %d caracteres %c\n"
          ,numero, carácter);
   system("pause");
   return 0;

 }//fin de la función main
```

1.9.5 Instrucciones de selección

Para implementar procesos de selección en un programa, los lenguajes C, C++. Java y C# utilizan

fundamentalmente dos instrucciones: la instrucción `if-else` y la instrucción `switch`. Ambas son muy utilizadas en los programas y se detallan en las siguientes unidades temáticas.

1.9.5.1 Codificación de if e if-else

La **codificación** de una estructura de selección puede tomar varias formas y éstas se codifican a través de la instrucción `if` o bien de `if-else`. La sintaxis general de la instrucción es la siguiente:

```
if (expresión) {
            Juego_de_Instrucciones_1;
}
[ else {
            Juego_de_Instrucciones_2;]
}
]
/* Nota: lo que se encuentra entre corche-
  tes[ ] significa que es código opcional,
  una instrucción puede llevar else o no
  llevarlo acorde a la estructura que se
  quiera programar */
```

En el código previo y en los posteriores de este capítulo

`Juego_de_Instrucciones_1`, `Juego_de_Instrucciones_2`, `Juego_de_Instrucciones_X` pueden representar una única instrucción o un grupo de instrucciones separadas por ";", `expresión` normalmente es una expresión booleana (devuelve `true` o `false`) o puede ser cualquier otra expresión válida u operando genérico dentro de la sintaxis del lenguaje (por ejemplo una expresión numérica en C).

El comportamiento de la instrucción if-else se puede resumir en los siguientes puntos:

- Si la expresión es verdadera (true) o diferente de cero (lenguaje C) se ejecutan la(s) instrucción(es) representadas por `Juego_de_Instrucciones _1`
- Si la expresión es falsa o igual a cero (lenguaje C)- no se ejecuta ninguna instrucción en caso de no existir `else`, o bien se ejecuta(n) la(s) instruccion(es) representadas por `Juego_de_Instrucciones_2`, que es el caso en donde se utiliza el `else`.
- Cuando se termina el proceso de ejecutar ya sea `Juego_de_Instrucciones 1`, `Juego_de_Instrucciones_2` o en el caso de que no se ejecute ninguna sentencia (cuando la expresión es falsa -o igual a cero- y no existe un `else`), el control del programa pasa a ejecutar la siguiente instrucción después de la instrucción `if-else`.
- expresión debe de ir entre paréntesis y cuando `Juego_de_Instrucciones_1` o `Juego_de_Instrucciones_2` es una única instrucción se pueden omitir las llaves de apertura "{"y de cierre "}", aunque muchos lenguajes sugieren por estilo que siempre se usen.

La relación entre los diagramas de flujo conteniendo estructuras de selección (figura 1.11) y su codificación correspondiente está dada en la figura 1.15.

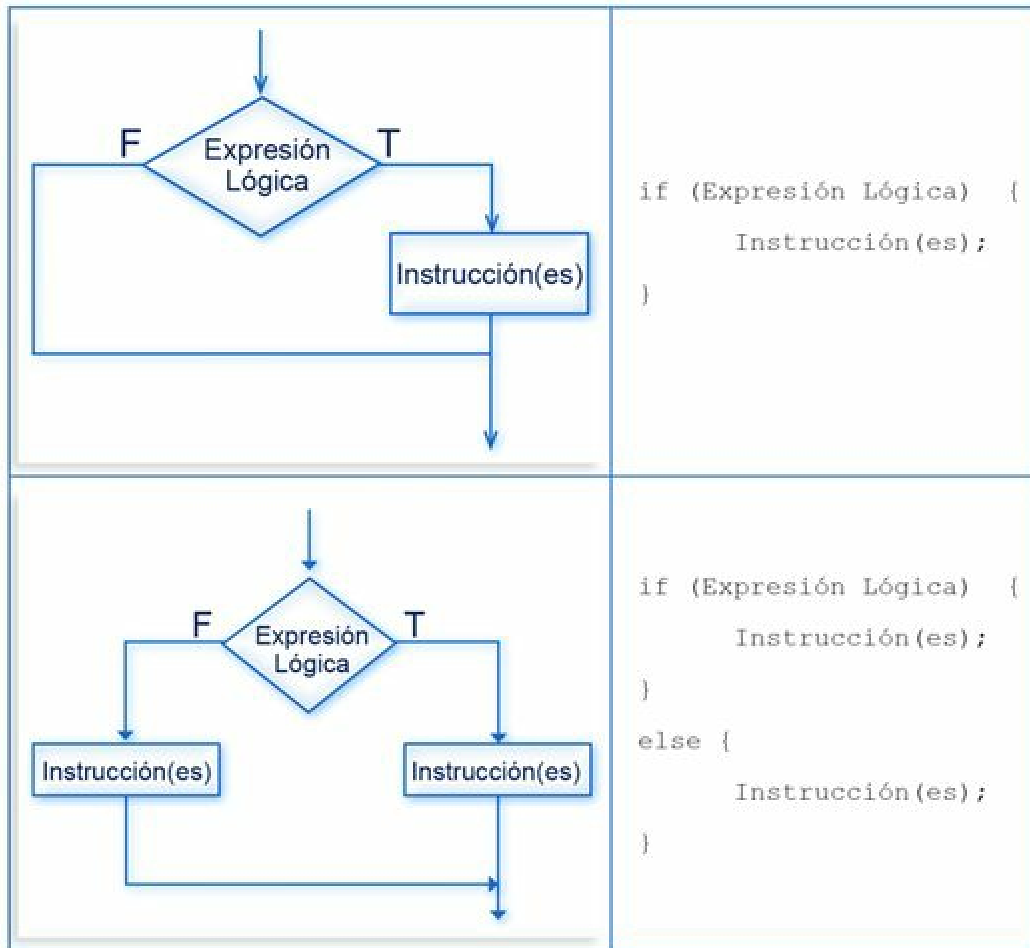


Figura 1.15. Relación entre estructuras de selección y código de programación.

1.9.5.2 Codificación de switch

En algunas ocasiones el uso de instrucciones `if-else` anidadas o secuenciales no es muy elegante en programación, ya que se pierde legibilidad en el programa. C, C++, Java y C# cuentan con la instrucción ***`switch`***, la cual es una versión más refinada de `if's` anidados. La sintaxis de la instrucción `switch` es la siguiente:

```
switch (expresión_discreta){
      case constante1: [Juego_de_Instrucciones_1;]
                       [break;]
      case constante2: [Juego_de_Instrucciones_2;]
                       [break;]
      case constante3: [Juego_de_Instrucciones_3;]
                       [break;]
                       .......
                       .......
      case constanten: [Juego_de_Instrucciones_n;]
                       [break;]
                       [default:  Juego_de_Instrucciones_default;]
}
/* Nota: lo que se encuentra entre corchetes[ ] significa que es código
opcional, una instrucción switch lo puede llevar o no acorde a lo que se
desee programar */
```

expresión_discreta es típicamente una expresión o variable tipo int o char. Las constantes (constante1, constante2... constanten) corresponden a valores posibles que pudiese tener expresión_discreta.

La operación de la sentencia `switch` se puede resumir en los siguientes puntos:

- Al inicio de la ejecución del `switch`, el programa busca el valor actual de `expresión_discreta` (dentro de la lista de constantes) y ejecuta el juego de instrucciones correspondientes; una vez hecho esto, ejecuta la instrucción ***`break`***, esta instrucción ignora el resto del código dentro de `switch` y hace que el control del programa proceda a ejecutar la siguiente sentencia después de la llave final "}" de la sentencia `switch`.
- El uso del ***`default`*** dentro de switch es opcional y se utiliza cuando se desea ejecutar una o más sentencias (`Juego_de_Instrucciones_default`) al no encontrar el valor que tiene actualmente la `expresión_discreta` en la lista de constantes.
- Cuando se desea ejecutar más de una instrucción dentro de uno de los casos del `switch` no se requiere agrupar éstas entre llaves ("{" y "}"), ya que la ejecución es secuencial hasta encontrar un `break`.
- Si no encuentra ninguna instrucción dentro de un caso de la instrucción switch (`Juego_de_Instrucciones_i` está vacío), y el valor de la expresión coincida con la constante correspondiente a este caso, se procede a ejecutar la sentencia `break`, si la hay y por lo tanto termina la instrucción `switch`, es decir no hace nada para este caso, si no hay instrucción `break` se procede a ejecutar Juego_de_Instrucciones_i+1, si éste no contiene instrucciones ejecuta el break si existe(por lo tanto termina la instrucción switch), y si no existe se pasa a ejecutar el juego de instrucciones del siguiente caso (`Juego_de_Instrucciones_i+2`) y así se continua con una ejecución secuencial hasta encontrar un `break` o finalizar la sentencia `switch`. De esta manera se pueden implementar listas de opciones con un mismo patrón de sentencias sin necesidad de repetirlas.

El siguiente código (incompleto) muestra un uso típico de la sentencia `switch`. Aquí se bosqueja una manera de implementar un programa tipo de altas, bajas, consultas y cambios de clientes.

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

int main()
{
     char opcion;
        printf("A.- Altas de clientes \n");
        printf("B.- Bajas de clientes \n");
        printf("C.- Cambios de clientes \n");
        printf("D.- Consultas de clientes \n");
        printf("S.- Salir del menú \n");

        printf("\n Introduzca su opcion: ");
        scanf("%c",&opcion);

        switch(opcion){
            case 'A':
            //Aquí irían las instrucciones correspondientes
            //a la alta de un cliente
                     break;
            case 'B':
           //Aquí irían las instrucciones correspondientes
           //a la baja de un cliente
                     break;
            case 'C':
           //Aquí irían las instrucciones correspondientes
           //a el cambio de un cliente
                     break;
            case 'D':
          //Aquí irían las instrucciones correspondientes
          //a las consultas de clientes
                     break;
            case 'S':
          //Aquí saldríamos del menú
                     break;
             default:
               printf("\n Opción no válida, ejecute de nuevo");
               printf("el programa");

        }//fin de la sentencia switch

        return 0;
}
```

Al código previo le hace falta la implementación de las acciones de altas, bajas, cambios y consultas, además de encapsularlo con una instrucción cíclica con el fin de que presentara un menú de manera repetitiva al usuario; después de cada acción ejecutada, estas **instrucciones cíclicas** se explican en la siguiente unidad temática.

1.9.6 Instrucciones de repetición

Como su nombre lo indica, las sentencias de repetición implementan estructuras cíclicas, las instrucciones de programación más comunes en este ámbito son: ***do-while***, ***while*** y ***for***.

1.9.6.1 Codificación de do-while

La sentencia `do-while` permite ejecutar un conjunto de instrucciones en forma cíclica mientras se cumple una condición. La sintaxis de la instrucción se muestra a continuación:

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

`Juego_de_Instrucciones` puede contener una o más sentencias separadas por “;” y `expresión` normalmente es una expresión booleana (devuelve `true` o `false`) o puede ser cualquier otra expresión válida dentro de la sintaxis del lenguaje (por ejemplo una expresión numérica en C).

La estructura de repetición asociada a la instrucción `do-while` y su codificación correspondiente está dada en la figura 1.16.

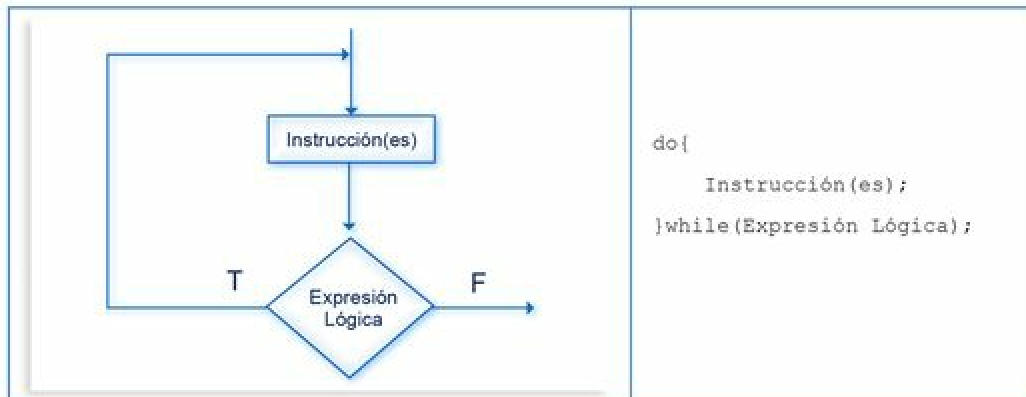


Figura 1.16. Relación entre la instrucción do-while y su estructura de repetición.

Las reglas de operación de la sentencia `do-while` son las siguientes:

- `expresión` debe llevar paréntesis y se evalúa al final del bloque de instrucciones, después de ejecutarse todas las sentencias.
- Si la expresión es verdadera (`true` ó diferente de cero en el caso de C), se repite el conjunto de instrucciones que encierra el `do-while`.
- Si la expresión es falsa (`false` ó cero en el caso de C), el control del programa se sale del ciclo y ejecuta la siguiente sentencia después del `do-while`.
- En caso de que sólo se desee ejecutar una sentencia dentro del ciclo, se pueden omitir las llaves (" {" y " }"), aunque en la mayoría de los lenguajes se sugiere su uso por cuestión de estilo.

Un ejemplo del uso de `do-while` se da en el siguiente código, aquí se complementa el código (aún incompleto) del sistema que da altas, bajas, cambios y consultas de clientes.

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

int main()
{
     char opcion;

     do{
        printf("A.- Altas de clientes \n");
        printf("B.- Bajas de clientes \n");
        printf("C.- Cambios de clientes \n");
        printf("D.- Consultas de clientes \n");
        printf("S.- Salir del menú \n");

        printf("\n Introduzca su opcion: ");
        scanf("%c",&opcion);

        switch(opcion){
            case 'A':
            //Aquí irían las instrucciones correspondientes
            //a la alta de un cliente
                     break;
            case 'B':
           //Aquí irían las instrucciones correspondientes
           //a la baja de un cliente
                     break;

           case 'C':
           //Aqui irían las instrucciones correspondientes
           //a el cambio de un cliente
                     break;
            case 'D':
           //Aquí irían las instrucciones correspondientes
           //a las consultas de clientes
                     break;
            case 'S':
            //Aquí saldríamos del menú
                     break;
             default:
               printf("\n Opción no válida..\n";

        }//fin de la sentencia switch
        } while(toupper(opción)!= 'S');

     return 0;

}
```

1.9.6.2 Codificación de while

Una sentencia muy parecida al `do-while` es la sentencia `while`; su sintaxis se muestra a continuación:

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

`Juego_de_Instrucciones` puede contener una o más sentencias separadas por ";" y `expresión` normalmente es una expresión booleana (devuelve `true` o `false`) o puede ser cualquier otra expresión válida dentro de la sintaxis del lenguaje (por ejemplo una expresión númerica en C).

La estructura de repetición asociada a la instrucción `while` y su codificación correspondiente está dada en la figura 1.17.

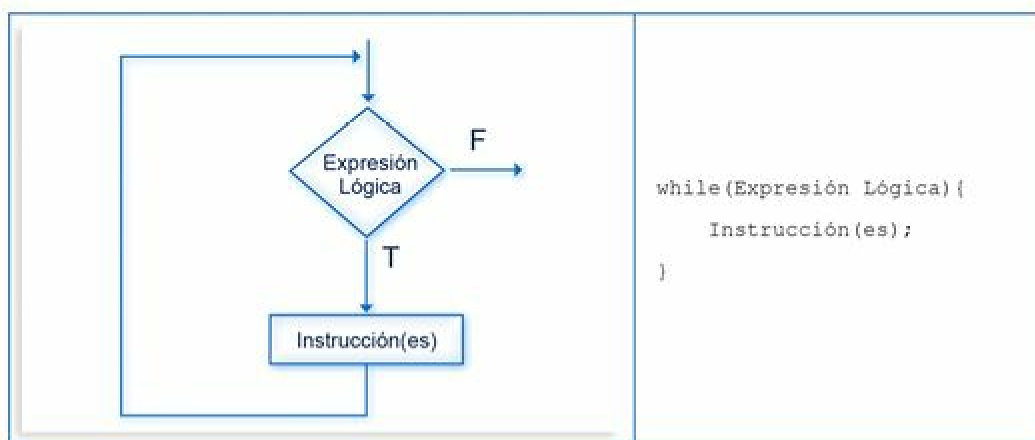


Figura 1.17. Relación entre la instrucción while y su estructura de repetición.

Como se muestra en las estructuras asociadas a `do-while` y `while`, la diferencia principal reside en que `while` efectúa la evaluación de la condición al principio de cada ciclo en vez de hacerlo al final, como lo hace `do-while`. Un ciclo controlado por `do-while` se ejecuta al menos una vez, un ciclo controlado por `while` puede no ejecutarse ninguna vez.

El comportamiento de la sentencia while se puede resumir en los siguientes puntos:

- `expresión` se evalúa al inicio del bloque de instrucciones la primera vez y después de ejecutarse todas las sentencias las veces siguientes.
- Si `expresión` es verdadera (`true` o diferente de 0 en el lenguaje C), se vuelve a repetir el conjunto de instrucciones encerradas en el ciclo. Al terminarse éstas se repite el proceso de evaluación como se mencionó en el punto anterior.
- Si `expresión` es falsa (`false` o igual a cero en el caso de C), se sale del ciclo y se ejecuta la siguiente sentencia después del final de la sentencia `while`.
- La sintaxis no requiere de llaves (" {" y " }") si dentro del ciclo existe sólo una sentencia, aunque en la mayoría de los lenguajes se sugiere su uso por cuestión de estilo.

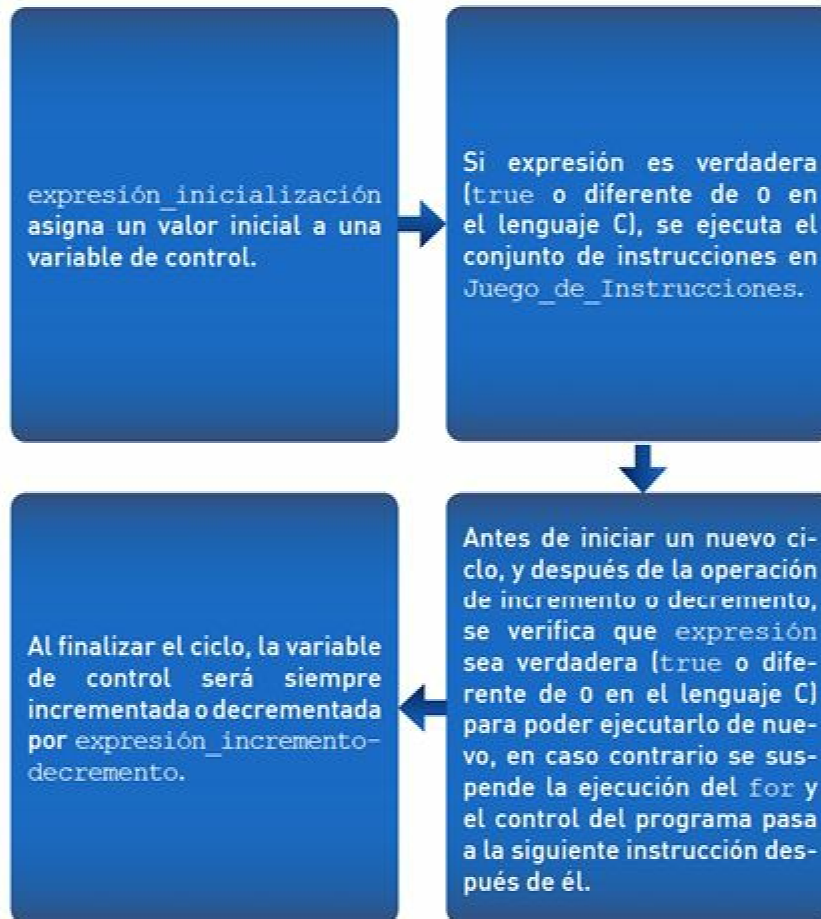
Una rutina de lectura de 10 números enteros positivos, indicando si son pares o impares utilizando una sentencia `while`, sería la siguiente:

```
cont=0;
while (cont<10){
        printf("dame un número positivo....");
        scanf("%d",&num);

        if (num%2==0) printf("es un número par\n");
        else printf("es un número impar\n");

        cont++;
}
```

1.9.6.3 Codificación de for

La sentencia `for` tiene una función similar al `while` y `do-while`, es decir, ejecutar un número de veces un conjunto de instrucciones. Se utiliza frecuentemente cuando se sabe cuántas veces hay que ejecutar un ciclo antes de entrar a él. La sintaxis del uso común de la sentencia es la siguiente:

```
for (expresión_inicialización; expresión;
                  expresión_incremento-decremento) {
                              Juego_de_Instrucciones;
}
```



La rutina de leer 10 números y desplegar si son pares o no se puede escribir con la sentencia `for` de la siguiente manera:

```
for (cont=1; cont<=10; cont++) {
     printf("dame un numero positivo....");
     scanf("%d",&num);

     if (num%2 == 0) printf("es un número par \n");
     else printf("es un número impar \n");
}
```

Nótese que para este tipo de casos es más sencillo y conveniente utilizar `for` en vez de `while` o `do-while`, ya que dentro del encabezado de la misma instrucción se controla el incremento de la variable `cont`.

La misma rutina pero decrementando la variable `cont` se muestra en el siguiente código:

```
for (cont=10; cont>0; cont--){
        printf("dame un numero positivo....");
        scanf("%d",&num);

        if (num%2==0) printf("es un numero par \n");
        else printf("es un numero impar \n");
}
```

En algunos programas es común utilizar la variable que se usa como contador para dar una mejor apariencia a la salida del programa, por ejemplo, si en el programa anterior el usuario desea saber qué número está alimentando (el primero, el segundo,.....el décimo) se podría hacer lo siguiente:

```
printf(" Se pediran 10 numeros y se indicara si son pares");
printf(" o impares\n");
for (cont=1; cont<=10; cont++){
       printf("%d.- dame un numero positivo....",cont);
       scanf("%d", &num);
       if (num%2 == 0) printf("es un número par \n");
       else printf("es un número impar \n");
}
```

con esto aparecería al inicio de la línea el valor de la variable cont.

La instrucción `for` es en realidad un ciclo `while`, ya que es equivalente a una sentencia `while` codificada de la siguiente manera:

```
expresión_inicialización;
while(expresión){
     instrucción_1;  // las instrucciones corresponden
     instrucción_2;  // a Juego_de_Instrucciones;
     ......          // en la sintaxis
     instrucción_n;
     expresión incremento-decremento;
}
```

Revisa la actividad al final del capítulo

1.10 Introducción a arreglos

En los lenguajes de programación una variable de un tipo sencillo (`float`, `int`, `char`, `double`, `boolean`) únicamente puede almacenar un dato a la vez, para almacenar varios datos del mismo tipo tenemos que declarar varias variables con diferentes nombres, esto último puede producir programas muy complicados de realizar si existe mucha información a almacenar.

Un arreglo es una solución inicial al problema de manejar altos volúmenes de información, podríamos definir a un **arreglo** de la siguiente manera:

> ***"Un arreglo es una variable que puede almacenar un conjunto de datos del mismo tipo a través de un único nombre"***

Un índice numérico asociado al nombre del arreglo indica cuál de los datos será accedido.

1.10.1 Arreglos unidimensionales (lenguaje C)

Los lenguajes de programación permiten arreglos de diferentes dimensiones, los arreglos más simples pero tal vez lo más utilizados son los arreglos de una dimensión o unidimensionales. Se verá en esta unidad temática cómo declarar y utilizar este tipo de arreglos.

Declaración de un arreglo unidimensional

La declaración de un arreglo de una dimensión tiene la siguiente sintaxis:

```
Tipo nombre[constante_entera];
```

donde Tipo es el tipo de dato a almacenar por el arreglo (double, float, int, char, etc.), nombre es el nombre de la variable que almacenará la información del arreglo y constante_entera es el número de elementos que contendrá el arreglo.

Si tenemos una declaración como:

```
int a[10];
```

Estamos en realidad declarando a la variable a como un arreglo de 10 elementos donde en cada uno de ellos se puede almacenar un número entero.

Arreglo a:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

La información almacenada en el arreglo puede ser accedida individualmente a través de un índice entre corchetes; el índice 0 corresponde al primer elemento del arreglo; el índice 1 al segundo; el índice 2 al tercero y así sucesivamente.

Para colocar información en un elemento del arreglo se puede utilizar una instrucción de asignación como la siguiente:

```
a[0] = -2;
a[7] = 15;
```

En el código previo se está almacenando un valor de -2 en la primera posición del arreglo y un valor de 15 a la octava posición.

Dependiendo del lenguaje y de la zona del programa donde está declarado, un arreglo puede inicializar todos sus elementos en 0 o no hacerlo. Debido a esto muchas veces se recomienda inicializar las variables y no asumir una inicialización implícita de parte del lenguaje.

Los elementos individuales de un arreglo pueden también ser utilizados dentro de una expresión, por ejemplo:

```
a[3] = 2*a[0] - a[7];
```

El código anterior almacenaría el valor de -19 en la cuarta posición del arreglo (considerando que previamente fueron almacenados los valores de -2 y 15 en a[0] y a[7]).

Expresiones como la siguiente también son permitidas y algunas veces utilizadas dentro del lenguaje C:

```
a[a[0]+3] = a[3] * a[a[7] -15];
```

Esto último almacenará en el elemento `a[1]` del arreglo el valor de 38 - considerando que ya fueron ejecutadas las instrucciones previamente mostradas- .

Así como en el ejemplo anterior se está almacenando un conjunto de enteros, es posible declarar arreglos de caracteres (esto se mostró en la unidad temática referente a *string*s), de floats, de doubles, etc. Por ejemplo, una declaración como la siguiente declararía un arreglo de 5 elementos tipo `double` y un arreglo de 12 elementos tipo `char`.

```
double f[5];
char s[12];
```

lo cual crearía en memoria dos arreglos:

Arreglo f:

f[0]	f[1]	f[2]	f[3]	f[4]

Arreglo `s`:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]	s[10]	s[11]

Un aspecto importante a mencionar es que para acceder un elemento de un arreglo debe de hacerse siempre a través de un índice o valor entero ya sea éste una constante o una variable. Por ejemplo, la siguiente rutina lee 5 números y los almacena en el arreglo de floats declarado

previamente.

```
int i; //se declara i como entero ya que la variable
       //será utilizada como un índice del arreglo

for(i=0; i<5; i++){
   printf("dame un número ");
   scanf("%f",&f[i]);
}
```

1.10.2 Arreglos bidimensionales (lenguaje C)

El lenguaje C, al igual que muchos lenguajes, permite manejar arreglos de varias dimensiones. Para declarar a una variable como un arreglo de varias dimensiones se hace de manera similar a como se declara un arreglo de una dimensión, solo que se agrega cuántos elementos tendrán posteriormente las siguientes dimensiones. La declaración de un arreglo de 2 dimensiones de datos tipo int es como sigue:

```
int b[2][4];    // se declara a la variable b como
                // un arreglo de dos dimensiones de enteros
                // tipo entero
```

Gráficamente podríamos representar al arreglo b una matriz de 2 renglones y de 4 columnas como se muestra en la siguiente gráfica.

Arreglo b:

b[0][0]	b[0][1]	b[0][2]	b[0][3]
b[1][0]	b[1][1]	b[1][2]	b[1][3]

Para este caso en particular, la variable b puede almacenar 8 valores enteros; estos elementos se pueden acceder a través de sus índices correspondientes. Nótese cómo el índice de cada dimensión inicia en cero.

Un arreglo puede ser inicializado al momento de su declaración, por ejemplo, las siguientes líneas de código

```
int a[10] = {10,34,-25,56,-46,1000,56,98,0,12};
int b[2][4] = {4,3,7,9,-2,45,980,12}:
```

inicializan el arreglo llamado a de la siguiente manera:

10	34	-25	56	-46	1000	56	98	0	12
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

e inicializan el arreglo llamado b de la siguiente manera:

b[0][0]	b[0][1]	b[0][2]	b[0][3]
4	3	7	9
-2	45	980	12
b[1][0]	b[1][1]	b[1][2]	b[1][3]

Para ejemplificar el uso de un arreglo de 2 dimensiones, en el apartado Recursos se muestra el código completo del juego del gato. Utiliza un arreglo de char's de 2 dimensiones, inicia con todas las posiciones conteniendo un espacio en blanco y, conforme va transcurriendo el juego, coloca en cada posición del arreglo la 'X' (usuario) o bien la 'O' (computadora), haz también uso de funciones auxiliares para simplificar la manera de resolver la problemática.

RECURSOS

Arreglo de 2 dimensiones – Juego del gato

Revisa la actividad al final del capítulo

1.11 Programación orientada a objetos

La programación orientada a objetos (OOP por sus siglas en inglés) es otro paradigma muy utilizado en programación, su enfoque es muy diferente a otro tipo de filosofías de programación y es muy útil en desarrollos de software complejos.

Mientras que la programación estructurada se enfoca principalmente en encontrar módulos (unidad temática 1.13) y algoritmos con estructuras adecuadas para solucionar problemáticas, la OOP se enfoca en resolver la problemática (encontrando los objetos que están inmersos en la situación), las relaciones y los mensajes existentes entre ellos. En OOP se modelan esos objetos a través de software y se implementan las relaciones y mensajes existentes entre ellos.

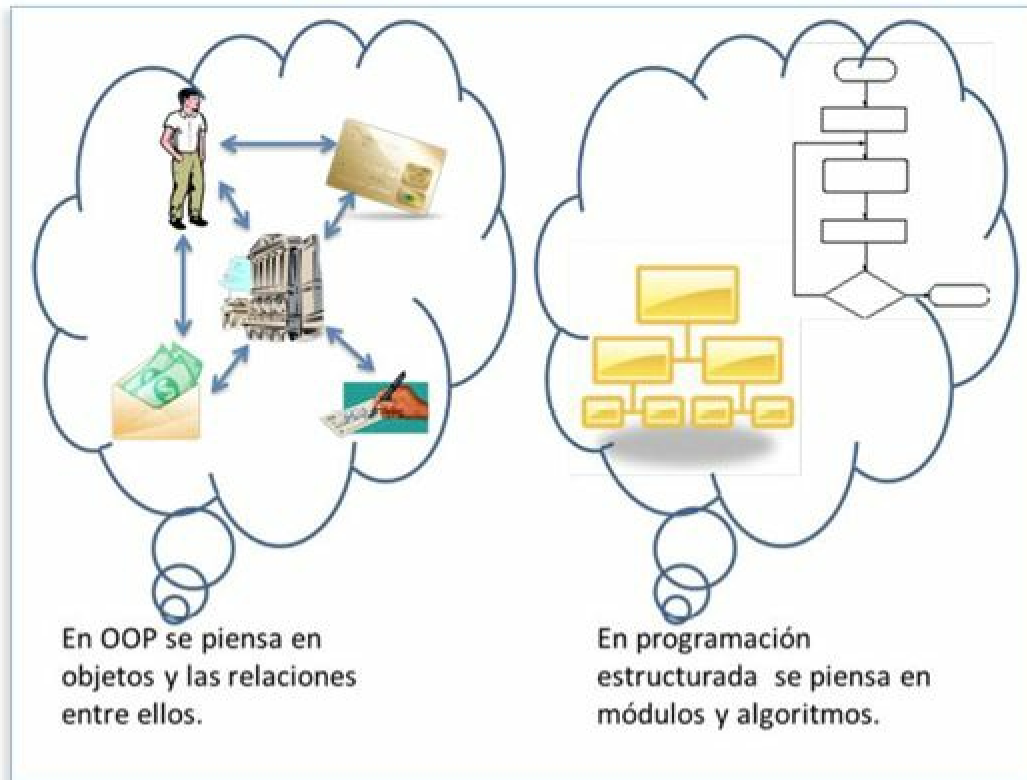


Figura 1.18. Programación orientada a objetos versus estructurada.

En un programa basado en OOP, las variables, aparte de almacenar datos simples (`int`, `float`, `double`, `char`, etc.), pueden almacenar objetos, los cuales son representaciones de *software*, que a su vez almacenan las propiedades o atributos del objeto y los mensajes a los que el objeto responde.

Tomemos el caso de modelar una cuenta bancaria para un sistema de *software* enfocado a bancos; una cuenta bancaria se puede ver conceptualmente como un objeto que contiene atributos como:

- el propietario
- el tipo de cuenta (cheques, ahorro, inversión, etc.)
- el saldo

Los mensajes o acciones que se pueden realizar con una cuenta bancaria y el nombre de la codificación propuesta en un lenguaje con filosofía OOP podrían ser:

Acción	*Codificación*
Consultar saldo	`saldo( )`
Depositar a la cuenta	`deposita(float n)`
Retirar de la cuenta	`retira(float n)`

Nota: a las variables declaradas en la codificación de un mensaje o acción se les llama parámetros, en este caso el parámetro `n` es de tipo `float` y sirve para indicar la cantidad de depósito o retiro de la cuenta al momento de realizar la acción.

Toda esta información (propiedades y acciones) se codifica en una entidad de *software* llamada clase, esta clase es como un molde en *software* que permite crear objetos con las características y comportamiento definidos en ella. Se pueden crear por lo tanto muchos objetos de una clase y almacenar cada objeto en una variable diferente.

Un ejemplo en el lenguaje Java para crear y manipular dos objetos de la clase mencionada anteriormente se hace de la siguiente manera:

```
1) CuentaBancaria c1,c2;
2) c1 = new CuentaBancaria("Juan Salas", "cheques", 1000.00);
3) c2 = new CuentaBancaria("Pedro Torres", "ahorro", 500.00);
4) c1.retira(300.00);
5) c2.deposita(400.00);
6) c1.retira(c1.saldo());
```

En el programa asumimos que `CuentaBancaria` es una clase existente dentro de la plataforma de Java en que estamos desarrollando el programa.

En la línea 1 se declaran dos variables que pueden almacenar objetos de la clase `CuentaBancaria` (hasta este momento no se han almacenado objetos en esas variables).

En la línea 2 se crea un objeto de la clase `CuentaBancaria` (a través del operador `new`) con propietario Juan Salas, de tipo cuenta de cheques, con un saldo inicial de 1000 pesos; este objeto se almacena en la variable `c1`.

En la línea 3 se crea otro objeto de la clase `CuentaBancaria` (a través del operador new) con propietario Pedro Torres, de tipo cuenta de ahorro, con un saldo inicial de 500 pesos; este objeto se almacena en la variable `c2`.

En la línea 4 se retiran 100 pesos de la cuenta bancaria que se almacena en la variable `c1` (en este momento el saldo de esa cuenta queda en 700 pesos).

En la línea 5 se depositan 400 pesos a la cuenta bancaria que se almacena en la variable `c2` (en este momento el saldo de esa cuenta queda en 900 pesos).

En la línea 6 se retira todo el saldo de la cuenta bancaria almacenada en `c1`.

Las acciones `saldo`, `deposita` y `retira` se codifican como funciones (en Java se llaman métodos) de la clase.

Existen también lenguajes que no siguen formalmente una filosofía particular sino más bien permiten mezclar en la programación diversos paradigmas (estructurado, objetos, modular, eventos, flujo de datos, etc.). Esto lleva muchas veces a producir programas que no son fácilmente escalables, reusables ni actualizados.

Cabe mencionar que la mayoría de los lenguajes modernos de programación (C++, Java, C#) están diseñados para trabajar en el paradigma OOP; sin embargo, pueden ser usados para soluciones con otros paradigmas, como el análisis estructurado y/o el análisis modular (el cual será descrito en la unidad temática 1.13)

Revisa la actividad al final del capítulo

1.12 Programación gráfica

Un tipo de programación que está cobrando mucho auge en el desarrollo de *software* en las áreas de ingeniería electrónica, ingeniería de control e instrumentación es la **programación gráfica**. Este tipo de programación está basada en el paradigma de flujo de datos, este paradigma define la ejecución de un programa a través de un flujo de información entre bloques o funciones que están conectados a través de alambrado. Un bloque únicamente puede ejecutarse si todas las entradas a él han sido recibidas a través del flujo del programa; la salida de este bloque puede llegar a las entradas de otros bloques en un programa, por lo que se crea un paralelismo innato en la ejecución. La figura 1.19 muestra este concepto en el lenguaje de LabVIEW. El bloque llamado “2D Pulse.vi” sólo puede ejecutarse y generar su salida si las cuatro entradas (array size, delay x, delay, width y) han llegado a su entrada; una vez generada la salida de este bloque, se procede a ejecutar en paralelo a los dos bloques llamados “FFT.vi” y la generación de la gráfica llamada “Image”, y así sucesivamente, de acuerdo a como va fluyendo la información en el programa.

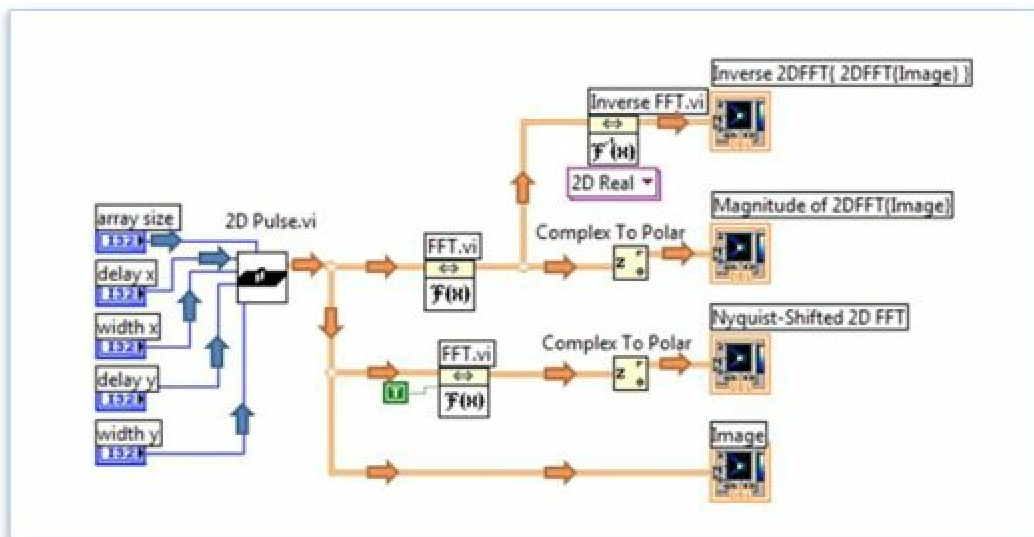


Figura 1.19. Programación gráfica, paradigma de flujo de datos.

LabVIEW[4] es el lenguaje representativo de este tipo de programación, fue introducido por la compañía National Instruments en 1986. Basado en el lenguaje de programación gráfico llamado G, ha permanecido como líder en desarrollo de soluciones en las mediciones de procesos de automatización y control.

1.12.1 Introducción a LabVIEW

LabVIEW es un ambiente de desarrollo basado en un lenguaje de programación gráfica para computadoras personales de diferentes plataformas, su propósito fundamental es desarrollar programas de instrumentación virtual, por eso a los programas en LabVIEW se les llama VIs (de Virtual Instruments por sus siglas en inglés).

Al tratarse de programación gráfica, no hay código escrito como lo existe en la mayoría de los lenguajes de programación, por lo que sus estructuras se representan de manera simbólica (aunque también existe la posibilidad de integrar estructuras de código escrito de lenguajes como C en los programas de LabVIEW). Una de las mayores ventajas de LabVIEW respecto al resto de los lenguajes es que facilita al usuario la comunicación con tarjetas conectadas a la PC y en general a cualquiera de los puertos de la computadora donde se ejecuta.

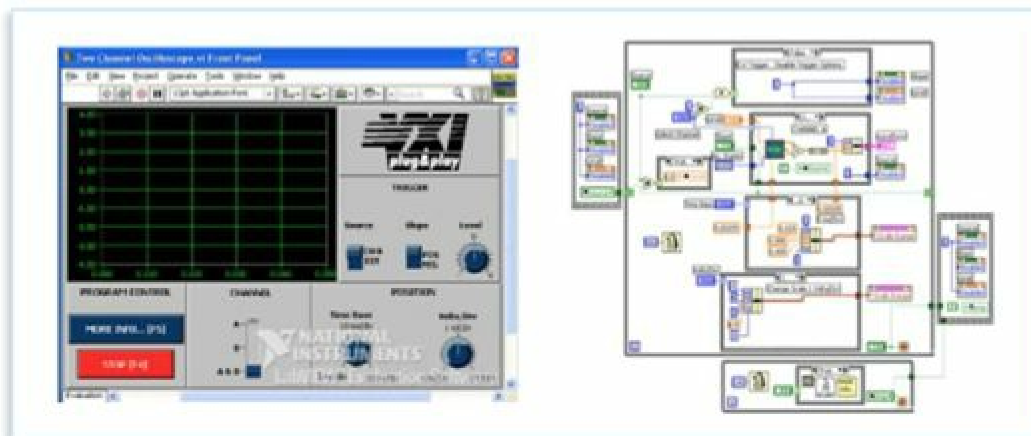


Figura 1.20. (a) Panel frontal y (b) diagrama de bloques (inferior) de un programa en LabVIEW.

LIGAS DE INTERÉS

Es la página oficial de LabVIEW, ahí puedes descargar la versión de evaluación del software, comprender muchas de las áreas de aplicación del mismo, foros de discusión, proyectos hechos en universidades con el software, etc.

LabVIEW

Cuando se desarrolla un programa en LabVIEW existen dos zonas de trabajo: la zona del panel frontal donde se colocan los controles que tendrá la interfaz del programa, y el *diagrama de bloques* que es donde se efectúa toda la programación "alambrando" los diferentes componentes y estructuras de un programa. En la figura 1.20 se muestra un panel frontal conteniendo controles de un **osciloscopio** virtual; en la misma figura (b) se muestra el alambrado correspondiente en el diagrama de bloques de dicha aplicación.

Panel frontal

Para colocar controles en el panel frontal se utiliza la barra de controles, esta se muestra haciendo clic derecho sobre este panel; en la figura 1.21 se muestra esta paleta. A través de ella se tiene entrada a controles agrupados por tipo dentro de otras subpaletas, en las que existe una diversidad de controles, indicadores, gráficas u objetos útiles; hay indicadores / controles de tipo numérico, booleano, *string*, además de formas predefinidas para colocar información de tablas, listas, arreglos, etc... A cada componente (control / interruptor / indicador / gráfica / etc.) ubicado en el panel frontal le corresponde un icono dentro del diagrama de bloques, de esta forma se asocia la interacción que ocurre en el panel frontal con la lógica de programación desarrollada en el alambrado hecho dentro del diagrama de bloques.

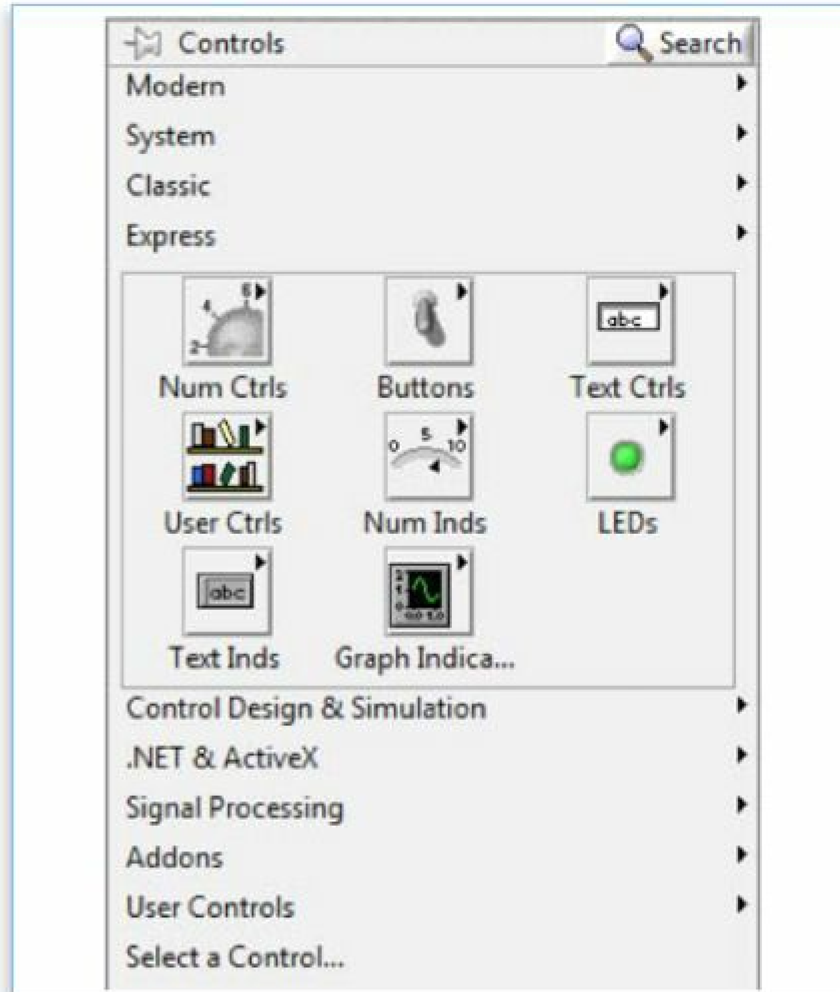


Figura 1.21 Barra de controles del panel frontal.

Diagrama de bloques

Las estructuras de programación más comunes con las que se trabaja en el diagrama de bloques en LabVIEW se encuentran en la paleta de Programación (*programming*) en la subpaleta de estructuras(structures), ésta es mostrada en la figura1.22 y se accede a ella con un clic derecho sobre el diagrama de bloques, en la ruta `Programming -> Structures`. Estructuras como while, for, case, etc., se encuentran en esta zona y son arrastradas al diagrama de bloques para desarrollar un programa.

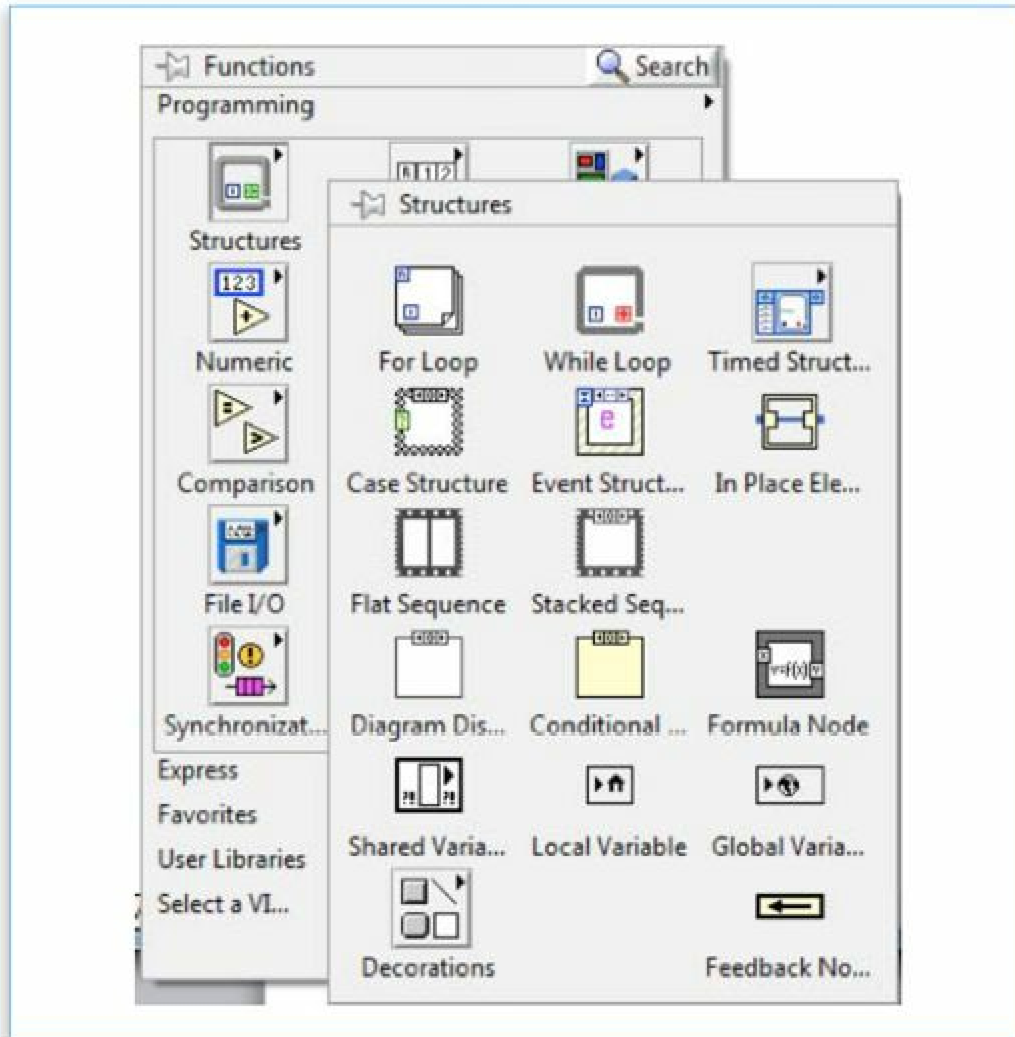


Figura 1.22 Paleta de programación (programming) en el diagrama de bloques.

Dentro de la mayoría de las estructuras pueden colocarse otras y un código de programación gráfico como se hace comúnmente en cualquier lenguaje de programación. En la figura 1.23 se muestra un grupo de paletas con diferentes tipos de bloques o funciones que pueden ser usados en la programación de una tarea dentro de LabVIEW.

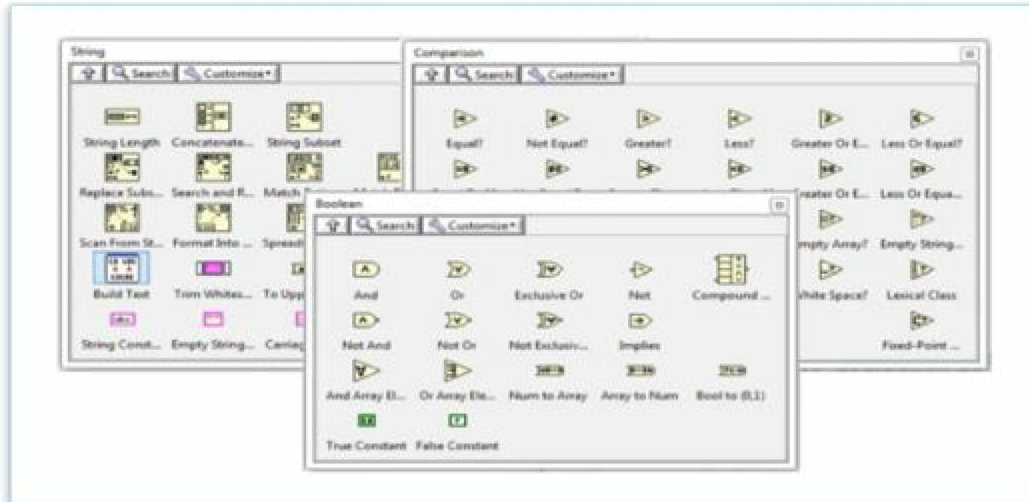


Figura 1.23 Paletas de String, Comparison y Boolean (diagrama de bloques).

Revisa los videotutoriales 1.1 y 1.2 que explican los fundamentos del entorno de programación de LabVIEW.

RECURSOS

[Video tutorial 1.1 - Introducción a LabVIEW Parte 1]

[Video tutorial 1.2 - Introducción a LabVIEW Parte 2]

1.12.2 Alimentando programas con código dentro de LabVIEW

Un recurso para solucionar problemáticas utilizado en LabVIEW programación de texto similar a C es la estructura Nodo de fórmula (*formula node*). A través de esta estructura se pueden alimentar datos de entrada y obtener datos de salida en función de estos datos de entrada. La estructura de nodo de fórmula se encuentra en la paleta de estructuras (*structures*) y se muestra su icono dentro de la figura 1.24.

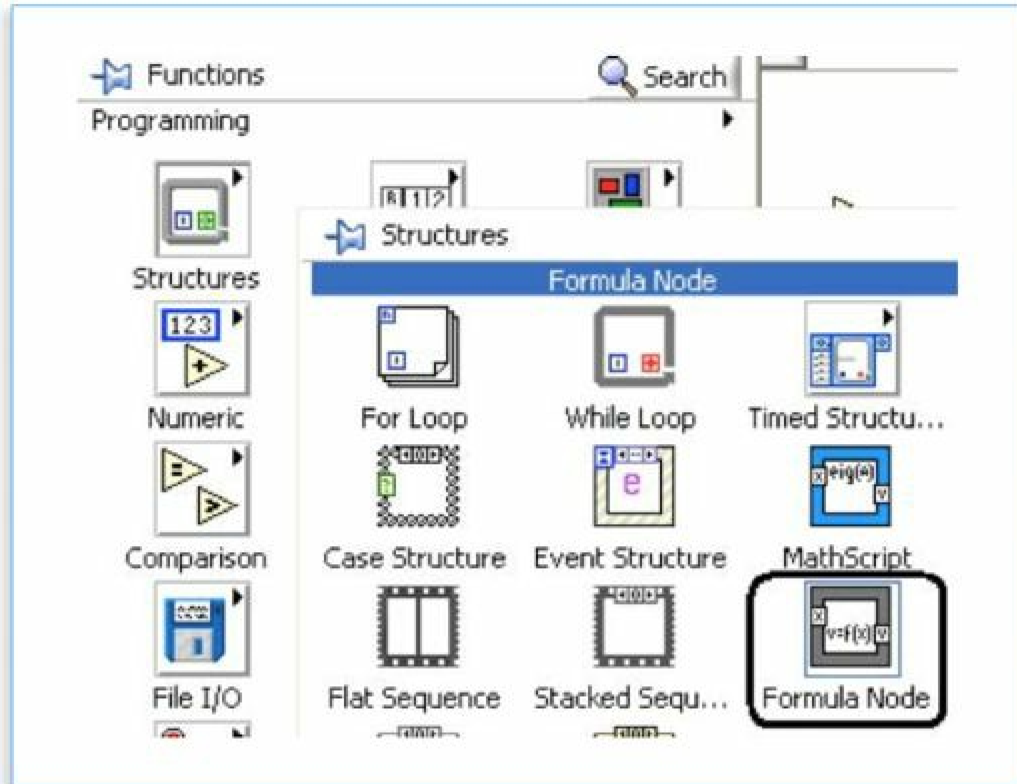


Figura 1.24 Acceso de la estructura nodo de fórmula.

Al utilizar una estructura nodo de fórmula se agregan entradas y salidas oprimiendo el botón derecho en el borde de ésta (ver figura 1.25). Las entradas se etiquetan con un nombre que puede usarse en la programación como si fuera una variable, las etiquetas de los nombres de salida deben corresponder con variables declaradas dentro del programa.

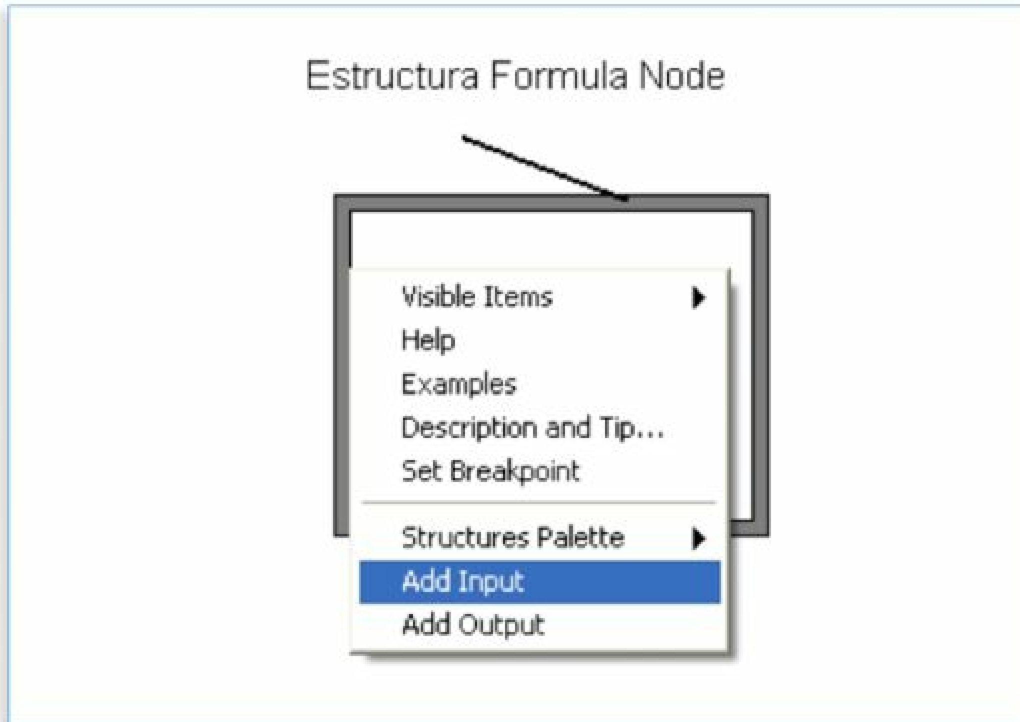


Figura 1.25 Añadiendo entradas y salidas al nodo de fórmula.

En la figura 1.26 se muestra un programa simple de LabVIEW que suma dos números; se muestra la solución utilizando alambrado común y la solución con nodo de fórmula.

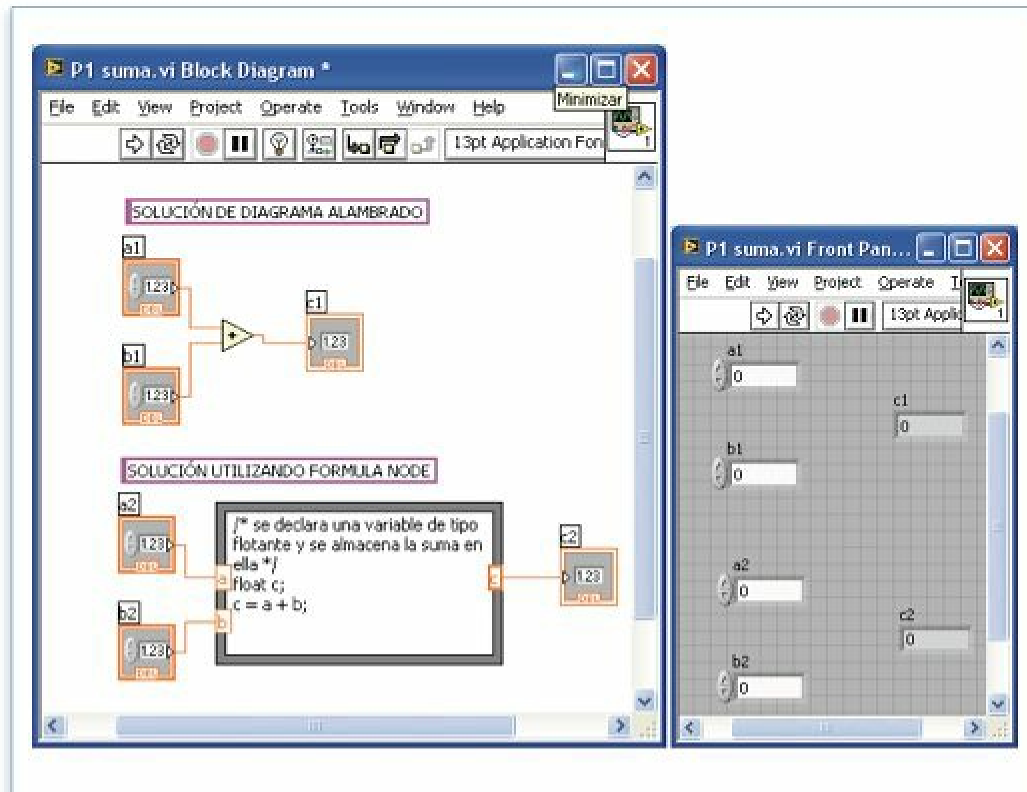


Figura 1.26 Programa en LabVIEW que suma dos números (modo gráfico y solución con nodo de fórmula).

En el apartado Recursos podrás revisar el videoturorial 1.3, en el cual, a través de un ejemplo se explica el manejo de la estructura del nodo de fórmula en LabVIEW.

RECURSOS

Video tutorial 1.3 – Estructura del modo de fórmula en LabVIEW.

Apéndice 1.2

Los tipos de datos que pueden ser usados dentro de la estructura *Formula Node* son: `int`, `int8`, `int16`, `int32` (enteros con signo), `uInt8`, `uInt16`, `uInt32` (enteros sin signo), `float`, `float32` y `float64` (punto flotante).

En el apéndice 1.2 se muestra la sintaxis completa del lenguaje o script que puede ser utilizado dentro del nodo de fórmula. Nótese como esta sintaxis es casi idéntica a la sintaxis de código en lenguaje C.

Revisa la actividad al final del capítulo

1.13 Programación modular

Una forma de solucionar de manera eficiente un problema grande en programación es dividirlo en problemas más pequeños y resolver de manera independiente cada uno de ellos. La **programación modular** tiene como filosofía o paradigma el crear subprogramas o módulos para resolver problemas pequeños de un problema mayor. La programación modular es una pieza fundamental de la programación estructurada; cuando se habla de desarrollo de software a través de programación estructurada, se está refiriendo también a la creación de unidades de programación (módulos) más manejables donde se utilicen estructuras secuenciales, de selección y repetición.

Las grandes ventajas que tiene un programador al solucionar un problema utilizando módulos son las siguientes:

» 1. Es más fácil visualizar y atacar un problema pequeño que uno grande.

» 2. El programa se vuelve más legible y entendible al utilizar módulos. Esto permite actualizar y depurar más rápidamente un programa.

» 3. El utilizar módulos en los lenguajes de alto nivel permite hacer un uso más eficiente de la memoria.

Aunque la mayoría de los lenguajes modernos de alto nivel no siguen el paradigma estructurado, la mayoría de ellos proveen mecanismos para realizar módulos dentro de sus programas.

Bajo esta propuesta se podría esquematizar el paradigma de la programación modular con un esquema como el de la figura 1.27.

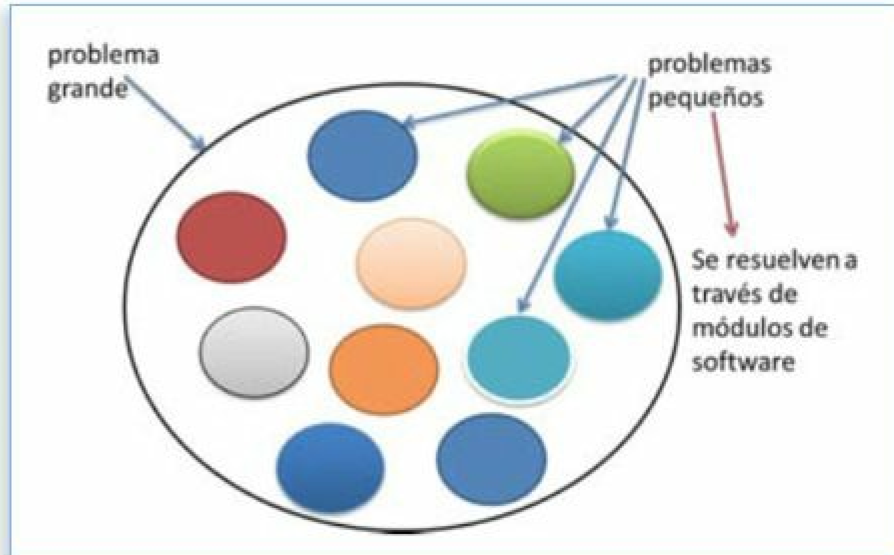


Figura 1.27 Paradigma de la programación modular.

En la mayoría de los lenguajes, los módulos se implementan a través de funciones / subrutinas / métodos, etc. Estos módulos de programación pueden clasificarse de manera general como:

Módulos que se enfocan en realizar una tarea o proceso dentro de un programa

Estos módulos pueden o no recibir datos de entrada para realizar la tarea / proceso para la que fueron diseñados. Esquemáticamente pudiesen representarse como lo muestra la figura 1.28.

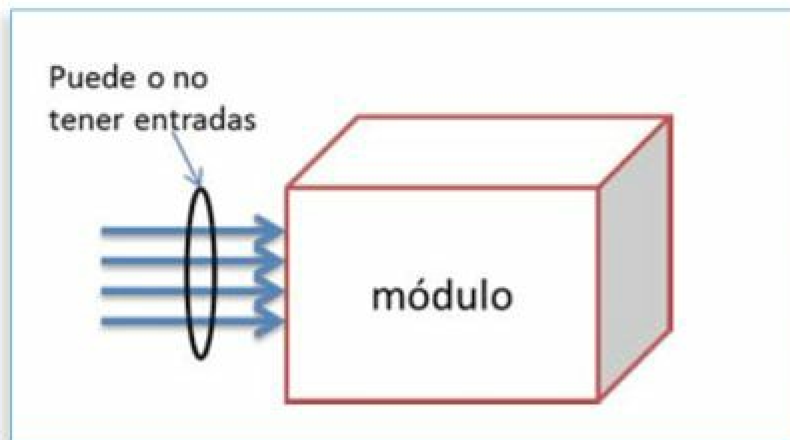


Figura 1.28 Módulos que realizan un proceso o tarea.

Procesos como estos podrían ser:

- "Limpiar" la pantalla de la computadora.
- Mover el cursor a una localización especifica de la pantalla.
- Dibujar un objeto o figura en la pantalla.

Módulos que se enfocan en obtener o calcular uno o más valores o datos útiles al programa

Estos módulos pueden o no recibir datos de entrada pero generan uno o varios valores de salida, esquemáticamente se podrían representar como lo muestra la figura 1.29.

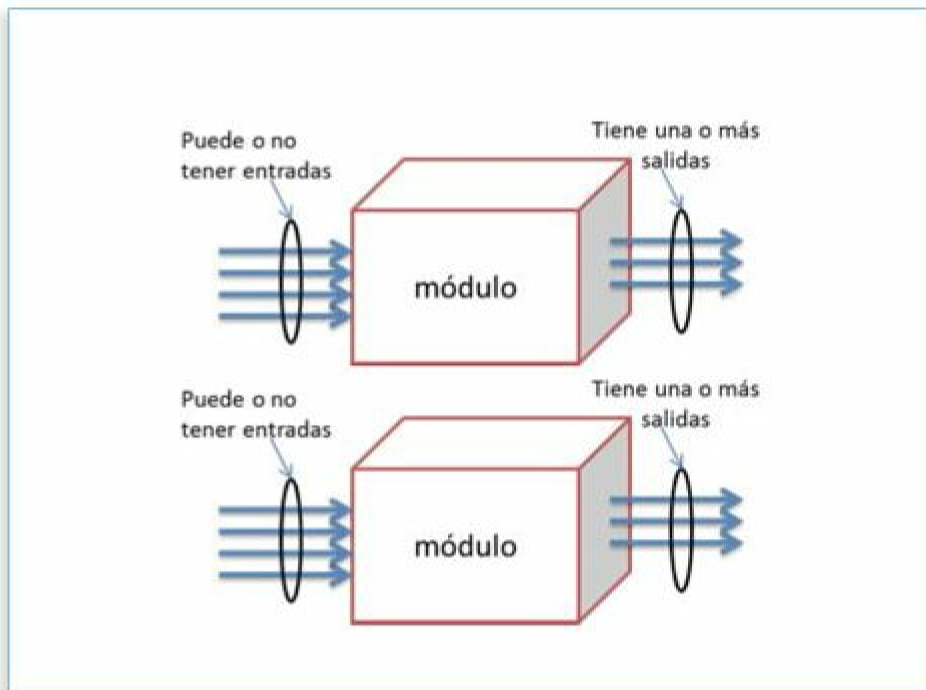


Figura 1.29 Módulos que obtienen uno o más datos útiles al programa.

Procesos como estos pudieran ser:

Obtener el promedio de los saldos de las diferentes cuentas de un usuario bancario.

Obtener el nombre del estudiante que tenga mayor promedio en un grupo.

Determinar las coordenadas de la posición actual del cursor del mouse.

Al igual que todos los **lenguajes de alto nivel**, LabVIEW tiene la capacidad de implementar módulos a través de lo que se llaman SubVIs que no son más que otros programas de LabVIEW que pueden ser insertados como módulos o bloques dentro de un programa o proyecto. La siguiente unidad temática trata de esto.

1.13.1 Implementando la programación modular a través de Labiew (SubVI´s)

LabVIEW puede implementar el concepto de programación modular integrando dentro de sus diagramas de bloques otros VIs (archivos con extensión vi) a través de una representación visual compacta. Con el fin de evitar un diagrama de bloques complejo, grande o demasiado alambrado se pueden identificar tareas que pueden ser encapsuladas a través de un módulo llamado dentro de LabVIEW subVI, con esto el programa se vuelve más legible y escalable. En realidad estos subVIs no son más que otros VIs con entradas y salidas claramente definidas que pueden ser utilizados dentro de cualquier programa de LabVIEW.

En el apartado Recursos podrás revisar el videoturorial 1.4, en el cual se muestra, a través de un ejemplo, cómo se aplica el concepto de programación modular dentro de LabVIEW.

RECURSOS
Video tutorial 1.4 – Programación modular dentro de LabVIEW.

Revisa la actividad al final del capítulo

Conclusión del capítulo 1

En este capítulo se ha dado un repaso intenso y amplio de los conceptos generales de programación, de las estructuras utilizadas en los diagramas de flujo cuando son usados para crear un algoritmo y de la sintaxis común de instrucciones típicas de programación. Como consecuencia de lo anterior el lector debe tener la habilidad de adaptarse a cualquier lenguaje de programación utilizando el paradigma estructurado.

También se introdujo al lector en el concepto de programación gráfica, esto en el ambiente de desarrollo de LabVIEW, con esto se tiene una visión temprana de gran parte de la programación que será utilizada en los capítulos posteriores de este eBook.

Las animaciones interactivas, ejercicios, videotutoriales y ejercicio integrador refuerzan los conceptos repasados en este capítulo y por ende la capacidad de análisis en el área de programación del estudiante que las realiza.

Actividades del capítulo 1

» Actividad 1.1

» Actividad 1.2

» Actividad 1.3

» Actividad 1.4

» Actividad 1.5

» Actividad 1.6

» Actividad 1.7

» Actividad 1.8 Parte 1

» Activdad 1.8 Parte 2

» Actividad 1.9 Parte 1

» Actividad 1.9 Parte 2

» Actividad 1.10

» Actividad 1.11

» Actividad 1.12

» Actividad 1.13

» Ejercicio integrador del capítulo 1. Parte 1

» Ejercicio integrador del capítulo 1. Parte 2

Recursos del capítulo 1

- Es la página oficial de LabVIEW, ahí puedes descargar la versión de evaluación del *software*, comprender muchas de las áreas de aplicación del mismo, foros de discusión, proyectos hechos en universidades con el *software*, etc.
  LabVIEW

- Es un sitio con mucha información respecto a C y C++, se requiere inscripción pero es gratuita. Ejemplos de programas en C y C++, foros de discusión, enlaces a otros sitios, etc., son sólo unos cuantos de los servicios que este sitio ofrece a programadores novatos, intermedios y avanzados.
  C Programming and C++ Programming

- Sitio oficial de Java, muestra todas las tecnologías alrededor de esta plataforma, tutoriales, descarga de la versión más nueva y su documentación, etc. Indispensable para aquellos que deseen introducirse en el mundo de la programación orientada a objetos.
  Java