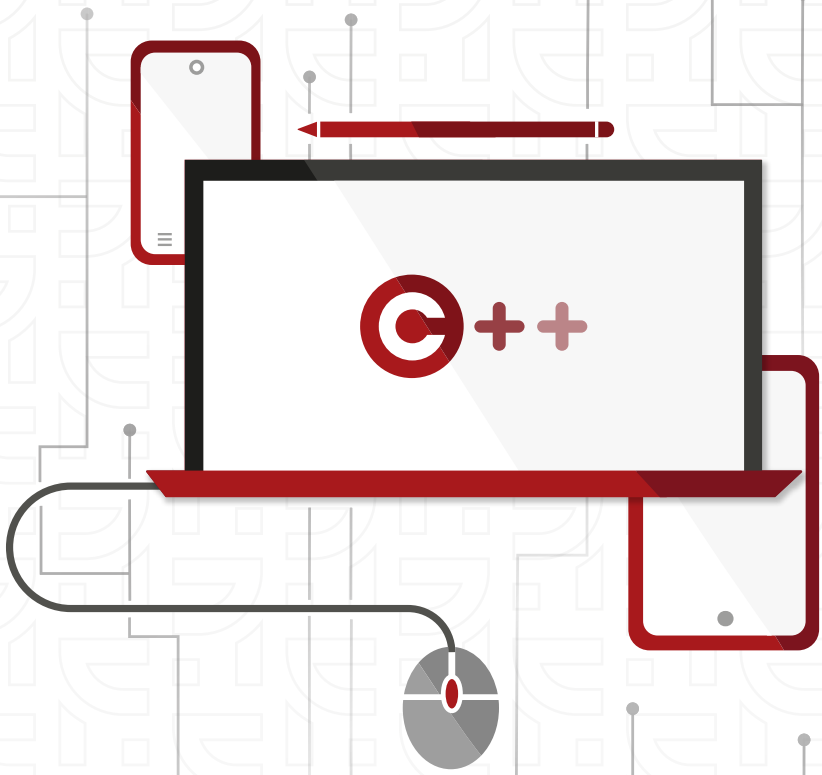
PROGRAMACIÓN DIGITAL

Juan Manuel Ramos Arreguín
Jesús Carlos Pedraza Ortega



UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

PROGRAMACIÓN DIGITAL

Dra. Silvia Lorena Amaya Llano
RECTORA

Dr. José Guadalupe Gómez Soto
SECRETARIO ACADÉMICO

Dr. Manuel Toledano Ayala
SECRETARIO DE INVESTIGACIÓN, INNOVACIÓN Y POSGRADO

Dra. María de la Luz Pérez Rea
DIRECTORA DE LA FACULTAD DE INGENIERÍA

Mtra. Diana Rodríguez Sánchez
DIRECTORA DEL FONDO EDITORIAL UNIVERSITARIO

Dr. Juan Carlos Jáuregui Correa
EDITOR RESPONSABLE

Mtro. Cristian Emanuel Tovar Navarro
COORDINADOR DEL DESPACHO DE PUBLICACIONES
FACULTAD DE INGENIERÍA

Mtro. Soid Lazlo Ruiz Ramírez
Andrea Cristina Garza Sandoval
CORRECCIÓN DE ESTILO

Lic. Mariana Cano León
DISEÑO EDITORIAL

Sarahí Velasco Hernández
DISEÑO DE PORTADA

Primera edición: 2026

ISBN del volumen: 978-607-513-788-9
ISBN de la colección: 978-607-513-787-2

La presente edición de *Programación digital* forma parte de la colección "Libros de texto" y es una publicación de la Facultad de Ingeniería de la Universidad Autónoma de Querétaro.

PROGRAMACIÓN DIGITAL

Juan Manuel Ramos Arreguín
Jesús Carlos Pedraza Ortega

Universidad Autónoma de Querétaro
Facultad de Ingeniería

DESPACHO DE PUBLICACIONES
FACULTAD DE INGENIERÍA

CONTENIDO

PRÓLOGO 11

1. INTRODUCCIÓN A LA PROGRAMACIÓN 14

1.1. Metodología de programación 15
1.1.1. Comprender el problema 16
1.1.2. Identificar y comprender la solución 16
1.1.3. Desarrollo del algoritmo 17
1.1.4. Desarrollo del diagrama de flujo 17
1.1.5. Documentación 18
1.2. Partes de un programa 18
1.3. Librerías en un programa en C++ 19
1.4. Comentarios 20
1.5. Definición de macros en C++ 20
1.6. Declaración de variables globales y locales 21
1.7. Declaración de funciones y subrutinas prototipo 22
1.8. Función principal: `main` 23

2. INTRODUCCIÓN A LA PROGRAMACIÓN EN C++ 26

2.1. Conceptos generales 26
2.1.1. Programación con lenguajes no estructurados 28
2.1.2. Programación con lenguajes estructurados 28
2.1.3. Programación orientada a objetos 29
2.2. Registros 29
2.3. Tipos de datos, variables y operadores 31
2.3.1. Declaración y asignación de variables 38
2.3.2. Asignar un valor 39
2.3.3. Valores con signo y sin signo 39
2.3.4. Variables tipo flotante de precisión sencilla IEEE 754 41
2.3.5. Variables tipo flotante de doble precisión IEEE 754 45
2.3.6. Error al definir un tipo de variable 48
2.3.7. Variables locales y globales 49
2.3.8. Operadores 53

– *Operadores de asignación* 54
– *Operadores aritméticos* 55
– *Operadores para apuntadores* 56
– *Operadores relacionales* 56
– *Operadores lógicos relacionales* 56
– *Condicionales* 60
– *Ejercicio: Identificar si un número es par o impar* 63
– *Ejercicio: Calcular las raíces de una ecuación cuadrática* 65
– *Decisión múltiple:* `(switch)` 69
2.3.9. Ciclos, repeticiones o bucles 74
– *Ciclo* `for` 74
– *Ejercicio: Imprimir los números del 0 al 9* 75
– *Ejercicio: Imprimir la tabla de multiplicar del 2* 78
– *Ejercicio: Imprimir los primeros N valores de la serie de Fibonacci* 80
– *Ejercicio: Ciclos anidados para imprimir las tablas del 1 al 5* 82
– *Ciclo* `do-while` 84
– *Ejercicio: Imprimir la tabla de multiplicar del 5 con ciclo* `do-while` 85
– *Ejercicio: Calcular el factorial de un número N* 87
– *Ciclo* `while` 90
– *Ejercicio: Tabla de multiplicar del número 7* 91
– *Diferencia entre los ciclos* 93
– *Comando* `break` 93
2.3.10. Constantes y macros 94
2.3.11. Arreglos y cadenas de caracteres 99
– *Arreglos unidimensionales* 99
– *Arreglos bidimensionales* 105
– *Errores comunes en el manejo de arreglos* 108
2.3.12. Apuntadores 112
– *Tamaño de variables* 112
– *Dirección de memoria* 113
– *Aplicación de los apuntadores* 115
– *Operaciones con apuntadores* 117
– *Uso de apuntadores con funciones* 118
– *Ejercicio: Usar apuntadores para convertir un valor decimal a binario* 119

2.3.13. Memoria dinámica en C++ 122
– *Función* `malloc()` 123
– *Función* `calloc()` 124
– *Función* `realloc()` 126
– *Función* `free()` 127
– `new` *y* `delete` 127
– *Ejercicio: Uso de arreglos con memoria dinámica* 128
2.3.14. Manejo de excepciones 130
– *Bloque* `try-catch()` *básico* 131
2.3.15. Sobrecarga de funciones 136
– *Ejercicio: Usar sobrecarga de funciones para calcular áreas* 136
– *Ejercicio: Usar sobrecarga de la función* `Potencia()` 137
2.3.16. Variables tipo `static` 138
2.4. Ejercicios de repaso 139

3. PROGRAMACIÓN ORIENTADA A OBJETOS 144

3.1. Estructuras 144
3.1.1. Inicializar valores de los campos 151
3.1.2. Funciones en una estructura 153
3.1.3. Uso de funciones prototipo en una estructura 156
3.1.4. Sobrecarga de funciones en una estructura 157
3.1.5. Función constructor 159
3.1.6. Función destructor 162
3.1.7. Sobrecarga del constructor 164
3.1.8. Asignación de estructuras 166
3.1.9. Arreglo de estructuras 168
3.1.10. Estructuras anidadas 169
3.1.11. Apuntadores a estructuras 170
3.1.12. Ejercicio: Operaciones con números complejos mediante estructuras 170
– *Comprensión del problema* 170
– *Algoritmo* 171
– *Diagrama de flujo* 172
– *Código en C++* 173
3.1.13. Sobrecarga de operadores para estructuras 174

3.1.14. Sobrecarga de operadores binarios para estructuras 175
3.1.15. Sobrecarga de operadores unarios para estructuras 181
3.2. Clases y miembros 183
3.2.1. Declaración de clases 183
3.2.2. Especificadores de acceso 184
3.2.3. Uso de las clases 184
3.2.4. Uso de funciones miembro 188
3.2.5. Uso del especificador de acceso `protected` 191
3.2.6. Constructor y destructor de clase 192
– *Constructor de una clase* 194
– *Sobrecarga del constructor* 198
– *Uso del destructor en clases* 202
3.2.7. Asignación inicial de datos a una clase usando listas 203
3.2.8. Arreglo de clases 205
3.2.9. Apuntadores a clases 208
3.2.10. El apuntador `this` 211
3.2.11. Clases derivadas 213
– *Aplicando herencia de clases* 217
– *Uso de constructores en clases derivadas o heredadas* 222
3.2.12. Polimorfismo y funciones virtuales 229
3.2.13. Sobrecarga de operadores para clases 232
– *Ejercicios: Sobrecarga de operadores* 238
3.2.14. Objetos plantilla 238
– *Funciones plantilla* 239
– *Clases plantilla* 241
3.2.15. Funciones amigas 244

REFERENCIAS **245**

ANEXOS **248**

4.1. Sistemas numéricos binario, octal y hexadecimal 248
4.2. Base numérica binaria 248
4.2.1. Conversión de decimal a binario para valores enteros 249
4.2.2. Conversión de decimal a binario para valores fraccionarios. 251
4.2.3. Conversión de binario a decimal 253

4.3. Base numérica octal 255
4.3.1. Conversión de decimal a octal para valores enteros 255
4.3.2. Conversión de decimal a octal para valores fraccionarios 257
4.3.3. Conversión de octal a decimal 258
4.4. Base numérica hexadecimal 259
4.4.1. Conversión de decimal a hexadecimal para valores enteros 260
4.4.2. Conversión de decimal a octal para valores fraccionarios 261
4.4.3. Conversión de hexadecimal a decimal 263

Prólogo

Prólogo

Programación digital es un libro de texto para la formación de nivel universitario. La directiva principal es satisfacer las necesidades educativas de los alumnos que cursan asignaturas como Programación, Programación Avanzada y, en menor medida, Microsistemas. En términos generales, está orientado a aquellos cursos que revisan y aplican el modelo de programación orientada a objetos.

La obra empieza por desentrañar cuestionamientos en torno al concepto de la programación: ¿de qué se trata?, ¿qué partes conforman un programa?, y ¿cómo puede uno iniciar con el desarrollo de programas? A su vez, se presenta una introducción a la programación en el lenguaje C++ donde se muestran los tipos de datos, variables y operadores, así como estructuras de control, sean condicionales, ciclos o arreglos. Asimismo, se exploran temas avanzados que abarcan apuntadores, memoria dinámica, excepciones y sobrecarga de funciones. En sucesión, se describe de modo amplio, pero no falto de detalle, el esquema de la programación orientada a objetos. Se parte de conceptos básicos, como estructuras, clases y miembros, listas y arreglos; al mismo tiempo ahonda temas como polimorfismo, apuntadores a clases, sobrecarga de operadores y objetos plantilla.

El contenido desplegado en este escrito se desarrolla como un curso pedagógico de programación digital. Se acompaña al estudiante en el proceso de aprendizaje de conceptos, a través de una guía estructurada, con el objetivo de fortalecer sus capacidades lógicas y analíticas para el diseño, el desarrollo y la implementación de algoritmos más complejos. Cada capítulo introduce desafíos progresivos e integra nuevos conceptos y conocimientos, proporcionando al mismo tiempo recursos que permitan elevar las habilidades de programación y perfeccionar las técnicas de diseño. Además, a lo largo de los apartados, se exhiben ejemplos de código C++, junto con algoritmos y diagramas de flujo que refuerzan los temas revisados para el desarrollo de programas más eficientes.

Para concluir, en los anexos se profundizan los sistemas numéricos binario, octal y hexadecimal, angulares en dispositivos digitales como microprocesadores o microcontroladores. Es imprescindible que los alumnos reconozcan tanto los sistemas de numeración como su relación con el decimal para su correcta aplicación en el desarrollo de programas digitales. Por ende, más que un simple compendio, *Programación digital* funge como una herramienta integral para el aprendizaje de la programación orientada a objetos. La presente capacita a los estudiantes universitarios en el diseño y desarrollo de algoritmos sofisticados, mediante su enfoque pedagógico y la amplia gama de ejemplos y desafíos. El propósito es que sus lectores dominen los conceptos expuestos, aprovechen los recursos proporcionados y adquieran las habilidades fundamentales en aras de enfrentar los desafíos de la programación digital, contribuyendo con soluciones innovadoras en el ámbito tecnológico.

Introducción a la programación

1. Introducción a la programación

Cuando se habla de programación, queda implícita una plataforma informática cuya funcionalidad ha sido preconfigurada para satisfacer alguna necesidad. Hoy en día, es innegable que el hombre está profundamente vinculado a los sistemas digitales, y peca de entender su funcionamiento sin requerir explicaciones, prescindiendo del profesional encargado de la programación correspondiente. Así pues, el hombre se ha convertido en un usuario arraigado a los dispositivos, tales como teléfonos, tabletas, televisores —inteligentes o digitales— sistemas computarizados para automóviles y aviones, por mencionar algunos. Empero, hoy en día resulta evidente la escasez de especialistas en programación de sistemas digitales. Una publicación de la Universidad de Guadalajara asevera que esta carestía representa un daño colateral para el desarrollo económico de México (UDG, 2021). A su vez, el periódico *El país* declaró que en 2020 se requería de 900 000 programadores (El país, 2021). A pesar de la demanda, la mayor parte de los especialistas se encuentran desempleados por diversas razones: conocimientos insuficientes, edad avanzada, habilidades nulas de promoción personal o reticencia a ampliar su aprendizaje técnico (Lores, 2001). En virtud de la constante evolución del mundo, el gremio debe mantenerse actualizado en el manejo de las tecnologías, en especial si el área de formación es programación de sistemas digitales.

Ahora bien, el sistema embebido es de particular relevancia para el ingeniero en automatización, mecatrónica, electrónica o cualquier rama ingenieril relacionada. Este dispositivo electrónico se distingue por su diseño, el cual le posibilita operar de forma autónoma, sin necesidad de un hardware adicional. En términos generales, los sistemas embebidos se clasifican en cuatro tipos:

1. **De uso comercial.** Esta categoría abarca dispositivos como teléfonos celulares, tabletas y televisores inteligentes. El hardware de estos equipos puede considerarse como

una "caja negra", ya que el usuario no puede modificarlo. Si bien es factible desarrollar soluciones de software para estos dispositivos, su diseño permanece inalterable. Tales sistemas dependen del uso de microprocesadores.

2. **De investigación y desarrollo tecnológico.** Se incluyen los dispositivos de computación de placa única (SBC), tales como Raspberry Pi, BeagleBoard y Odroid, cuyo objetivo es servir como plataformas versátiles para el desarrollo de software enfocado en la solución de diferentes problemáticas; los más usuales corresponden a tareas de vigilancia, detección de rostros, procesamiento de imágenes y robótica. En contraste con los sistemas de uso comercial, estos pueden ser integrados con hardware externo para ampliar sus capacidades.
3. **Microcontroladores.** Los circuitos integrados programables se consideran sistemas embebidos, ya que su diseño cuenta con lo necesario para operar de manera autónoma. A diferencia de otros sistemas más prototípicos, los microcontroladores deben programarse antes de su uso. Estos dispositivos se emplean para ejecutar tareas específicas, en lugar de procesar grandes volúmenes de información. Asimismo, ofrecen al usuario la flexibilidad de desarrollar su propio sistema embebido a partir de los microcontroladores.
4. **Sistemas en un solo encapsulado.** También denominados System on a Chip (SoC), estos sistemas se desarrollan con base en dispositivos lógicos programables, como los CPLD o FPGA. Los SoC permiten crear un sistema completo y encapsularlo todo en un solo circuito integrado.

Este libro se enfoca en los lenguajes de programación C y C++ para computadora, en aras de familiarizar al lector con la programación de microcontroladores a través de un análisis contrastivo entre ambos.

1.1. METODOLOGÍA DE PROGRAMACIÓN

En la Figura 1.1 se presenta un diagrama esquemático del proceso que debe llevarse a cabo cuando se trata de desarrollar un programa de computadora. Al respecto, es necesario aclarar que el diagrama opera bajo la suposición de que tanto la PC como los sistemas embebidos con su propio sistema operativo disponen de suficiente capacidad de almacenamiento para ejecutar todas las funciones que se les asignen.

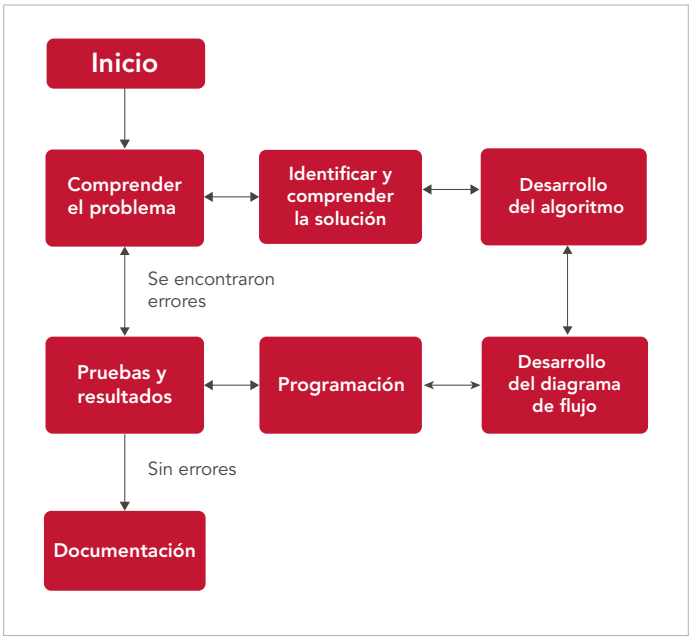


Figura 1.1. Metodología de programación.

Se recomienda que el estudiante siga esta metodología, planifique el desarrollo de soluciones y, particularmente, adopte un enfoque estructurado para la creación de programas. A continuación, se describe la importancia que tiene cada bloque de la Figura 1.1.

1.1.1. COMPRENDER EL PROBLEMA

Es improbable encontrar una solución cuando el camino hacia ella es incierto. Por lo tanto, para desarrollar un programa que resuelva un problema, es imperativa una absoluta comprensión de los obstáculos a superar.

1.1.2. IDENTIFICAR Y COMPRENDER LA SOLUCIÓN

Una vez que el problema ha quedado definido, se debe identificar la forma de resolverlo. Empero, no basta con entender el proceso para solucionarlo, es imprescindible comprender el método por el cual se desarrolla dicha solución. Es decir, dado que la solución consiste en

la aplicación de algún algoritmo o método matemático, es indispensable vislumbrarlos por medio de una serie de ejercicios que ayuden a comparar los resultados del programa y saber si funcionan correctamente o no.

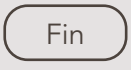
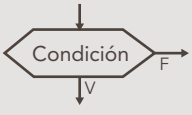
1.1.3. DESARROLLO DEL ALGORITMO

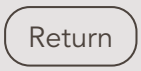
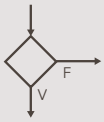
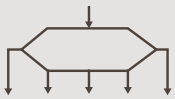
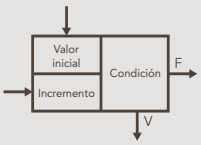
Esta etapa es esencial en el desarrollo de un programa. Antes de escribir cualquier código es crucial crear un algoritmo que brinde una visión clara de los pasos a seguir en la programación de la solución. Para decantarse por un camino, en primera instancia es necesario considerar todas las alternativas que puedan seguirse, haciendo alusión a, por lo menos, un caso de estudio como referencia.

1.1.4. DESARROLLO DEL DIAGRAMA DE FLUJO

En la Tabla 1.1 se presentan los bloques que conforman un diagrama de flujo. Cada uno de ellos desempeña una función específica al representar de forma gráfica la secuencia de pasos a seguir en un procedimiento.

Tabla 1.1. Bloques disponibles en el diseño de un diagrama de flujo.

SÍMBOLO	SIGNIFICADO	SÍMBOLO	SIGNIFICADO
	Flechas de sentido de flujo.		Conector de flujo de datos.
	Inicio de un diagrama de flujo.		Bloque para solicitud de información.
	Inicio de una función o subrutina.		Bloque de asignación.
	Fin del diagrama de flujo.		Bloque condicional 1. Modifica el flujo de datos en el diagrama de flujo.

SÍMBOLO	SIGNIFICADO	SÍMBOLO	SIGNIFICADO
	Retorno o fin de una subrutina o función.		Bloque condicional 2. Modifica el flujo de datos en el diagrama de flujo.
	Bloque de llamado a una función o subrutina.		Bloque condicional de salida múltiple.
	Bloque para impresión de datos o mensajes.		Ciclo *for*.

1.1.5. DOCUMENTACIÓN

Cuando el programa esté libre de errores y se haya verificado que la solución es correcta, se procederá a documentar el código. Para ello es necesario acatar alguna de las normativas vigentes o adoptar una metodología particular.

1.2. PARTES DE UN PROGRAMA

Todo programa consta de segmentos esenciales y no esenciales. En la Figura 1.2 se muestra un ejemplo de ambos tipos. Entre los elementos imprescindibles se incluyen las librerías, la rutina principal y la declaración de variables locales y globales. Por otro lado, las secciones prescindibles del programa comprenden subrutinas, funciones, interrupciones y macros.

Las variables se clasifican en dos grupos: en primer lugar, las locales, que se declaran dentro de funciones o subrutinas y son válidas siempre que el flujo del programa permanezca dentro de esos dominios; en segundo, las globales, que tienen vigencia en cualquier parte del programa, independientemente de la ubicación del flujo. En lo sucesivo, se proporciona una breve explicación de estos conceptos, centrada en un código en C++.

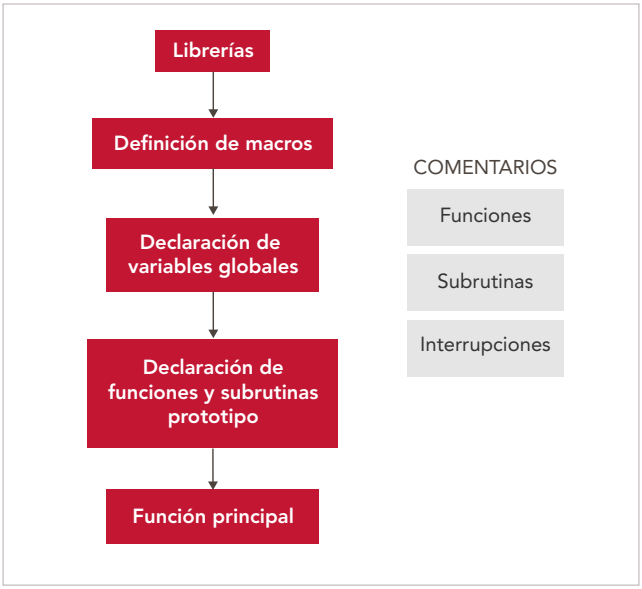


Figura 1.2. Partes de un programa.

1.3. LIBRERÍAS EN UN PROGRAMA EN C++

Las librerías son colecciones de funciones básicas requeridas para el desarrollo de programas en C++. Entre las principales bibliotecas en este lenguaje de programación se contemplan las siguientes:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream> (Esta no requiere los archivos  stdio.h  y  stdlib.h)
```

Existen diversas librerías disponibles, cada una con un conjunto único de subrutinas y procedimientos diseñadas para realizar tareas específicas. Se recomienda familiarizarse con las funciones de cada librería y su aplicabilidad; sitios web como *CPlusPlus.com* (https://www.cplusplus.com/) ofrecen una amplia gama de recursos e información para profundizar en su utilización.

1.4. COMENTARIOS

Los comentarios constituyen una herramienta para documentar un programa, ya que ayudan a los programadores a recordar todo tipo de detalles dentro del código, desde el propósito de cada sección hasta la función de cada variable. Como consecuencia, existen dos tipos principales de comentarios: por resto de línea o por bloque.

Para insertar comentarios por resto de línea, se emplea la doble diagonal "`//`", símbolo que indica que el compilador ignorará todo lo que se escriba a continuación en el mismo renglón. En cambio, para añadir un comentario por bloque, se utiliza "`/*`" al inicio y "`*/`" al final, lo cual permite al compilador omitir todo el texto que se encuentre entre estos acotadores. El Código 1.1 ejemplifica el uso de ambos tipos de comentarios.

1.5. DEFINICIÓN DE MACROS EN C++

Las macros en C++ son construcciones que permiten definir una secuencia de código que se expande antes de que se compile el programa. Se emplean para establecer constantes y funciones, las primeras de las cuales se caracterizan por la imposibilidad de cambiar su valor en la ejecución del programa, y deben asociarse a un nombre que cumpla con las mismas reglas de definición de una variable. A continuación se presenta un ejemplo de definición de constante donde se asigna el valor 3.14159 a la variable π.

```
#define pi 3.14159
```

Como se mencionó, las macros también permiten definir funciones. Por ejemplo, en la siguiente línea se define la función $f(A,B) = \frac{-B}{2A}$

```
#define f2(A,B)   -B/(2*A)
```

Estos procedimientos se explican de forma más detallada en el capítulo 2, apartado "Constantes y macros", página 94.

1.6. DECLARACIÓN DE VARIABLES GLOBALES Y LOCALES

Entre las partes que configuran un programa se encuentran las variables globales y locales. Las globales se caracterizan por gozar de validez en cualquier parte del código y, por lo general, se declaran después de las librerías y los macros. En el Código 1.1 se muestran ejemplos de declaración de variables globales y locales. En el caso de las globales, aunque pueden declararse al inicio de la función principal (`main`) o en cualquier otro instante, se recomienda hacerlo al comienzo del `main` como una buena práctica de programación. Aun así, pueden definirse en cualquier sección del programa, siempre que se ubiquen fuera de una función o subrutina. Por el contrario, las locales solo disponen de alcance dentro de la función o bloque de instrucciones donde han sido definidas.

Código 1.1. Ejemplo de declaración de variables locales y globales.

```
//Librerias a usar
#include <stdio.h>
#include <stdlib.h>

//Definición de macros
#define pi 3.14159

//Declaración de las variables globales, válidas en cualquier parte del
programa
int j, k;
char a1, a2;
float b;

void main()
{
       //Decalaración de variables locales
       int o, c;
       double d;

       for (int i = 0; i < 5; i++)
   {
        /*La variable local i es válida
          solamente en este bloque de instrucciones*/
        printf("%d, ", i);
   }
       //La variable local i ya fue destruida
}
```

1.7. DECLARACIÓN DE FUNCIONES Y SUBRUTINAS PROTOTIPO

Es preciso establecer categóricamente la diferencia entre una función y una subrutina. La función es un bloque de código con un nombre específico que, al ejecutarse, entrega o retorna un valor concreto. En cambio, la subrutina también corresponde a un bloque de código, empero no devuelve valor alguno. Para evitar errores de compilación, es menester declarar previamente las funciones y subrutinas que se utilizarán en un programa. De lo contrario el compilador marcará error de sintaxis si encuentra el llamado a una función o subrutina cuyo código aún no haya sido compilado de antemano. El Código 1.2 ilustra la forma correcta de declarar tanto funciones como subrutinas.

Código 1.2. Ejemplo de definición de prototipos de subrutina y función.

```
//Librerias a usar
#include <stdio.h>
#include <stdlib.h>
//Definición de macros
#define pi 3.14159

//definición de prototipos
void funcion1();
int funcion2();

//Declaración de las variables globales
int j;

void main()
{
	funcion1();
	j = funcion2();
}

void funcion1()
{
	//Aquí se colocan las instrucciones necesarias
}

int funcion2()
{
	//Bloque de instrucciones
	Return /*valor entero*/
}
```

Cuando una función es empleada como subrutina, el prototipo debe declararse como tipo `void`. De manera alternativa, si se declara como cualquier otro tipo (`int`, `float` o `double`), implica que la función debe retornar un valor del tipo especificado en el prototipo.

1.8. FUNCIÓN PRINCIPAL: `main`

Toda aplicación desarrollada en C/C++ debe incluir el comando `main`, este es el primer punto de entrada al ejecutar el programa. Es capaz de operar como función o subrutina, y tiene la opción de recibir o no argumentos. El Código 1.3 muestra un programa donde se usa el `main` sin tipo ni argumentos. Mientras el Código 1.4 revela un programa donde el método funciona sin parámetros y como una función de tipo flotante, lo que implica retornar a un valor flotante. Por último, el Código 1.5 refiere al `main` sin tipo pero con valores de entrada: la variable `argc` recibe el número de entradas y el arreglo `argv` almacena los datos adicionales que se envían al programa.

Código 1.3. `main` sin tipo ni argumentos.

```
//Encabezados

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

void main()
{
	/*
	* Bloque de instrucciones
	*/
}
```

Código 1.5. `main` de tipo flotante sin argumentos.

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

float main()
```

```
{
      /*
      * Bloque de instrucciones
      */
      //Se retorna el valor de 1.5 flotante
       return 1.5f;
}
```

Código 1.6. `main` sin tipo y con argumentos.

```
//Encabezados
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void main(int argc, char *argv[])
{
      int i;

      printf("El número de argumentos es: %d\n", argc);
      for (i = 0; i < argc; i++)
   printf("Argumento número %d: %s\n", i, argv[i]);
      _getch();
}
```

Es relevante considerar que se utiliza el comando **`_getch()`** en lugar de **`getch()`**, ya que el entorno Visual Studio —donde se desarrollan estos ejemplos— plantea el primero como un comando inseguro, motivo por el cual lo clasifica como un error. Una vez compilado, el programa debe ejecutarse desde la ventana de comandos de Windows. Suponiendo que el nombre asignado al proyecto sea "**`ProgramaMain`**", el comando correspondiente sería:

```
ProgramaMain antes ahora despues
```

Y la salida que se obtendría es la siguiente:

```
El número de argumentos es: 4
Argumento número 0: ProgramaMain
Argumento número 1: antes
Argumento número 2: ahora
Argumento número 3: despues
```

Aquí se concluye la primera sección. Una vez adquirida una comprensión general sobre la programación, en el siguiente apartado se da inicio a una introducción al lenguaje C++.

Introducción a la programación en C++

2. Introducción a la programación en C++

2.1. CONCEPTOS GENERALES

El lenguaje de programación ha impactado profundamente en la vida cotidiana. Su alcance se extiende desde rubros industriales y de producción, hasta contextos educativos, domésticos, recreativos y de entretenimiento. A simple vista su presencia pasa inadvertida; no obstante, está detrás del funcionamiento de dispositivos cotidianos, tales como reproductores de video, electrodomésticos, teléfonos móviles, televisores inteligentes, videojuegos, procesadores de texto y un largo etcétera. Sin embargo, los lenguajes de programación han recorrido un largo camino en su evolución y desarrollo, como se muestra en la Tabla 2.1.

Tabla 2.1. Evolución de los lenguajes de programación.

AÑO	SOFTWARE	AÑO	SOFTWARE
1843	Primer lenguaje de programación	1986	Erlang, Objective – C
1940	Lenguaje máquina	1987	Perl
1950	Lenguaje ensamblador	1988	Mathematica
1952	Autocoder	1989	FL
1954	IPL	1990	Haskell
1955	Flow Matic	1991	Python, Visual Basic, HTML
1957	Fortran	1993	Ruby, Lua
1958	LISP	1994	CLOS

1959	FACT, COBOL, RPG	1995	Java, JavaScript, PHP, Delphi (Object Pascal)
1962	APL, Simula y SNOBOL	1996	WebDNA
1963	CPL, Algol	1997	Rebol, Lenguaje de programación R
1964	Basic	1998	Erlang
1967	BCPL	1999	D
1968	Logo	2000	ActionScript
1969	B (Previo a C)	2001	C#, Visual Basic.NET, Lava
1970	Pascal, Forth	2002	F#
1972	C, Prolog, Smalltalk	2003	Groovy, Scala, Factor
1973	ML	2005	Scratch, Ruby On Rails
1975	Scheme	2007	Clojure
1978	SQL	2009	Go
1980	Ada, Objective-C	2010	Kotlin, Rust
1983	C++	2011	Dart
1984	Matlab, Common Lisp, Postscript	2012	TypeScript, Elixir
1985	Eiffel	2014	Swift

El desarrollo de los lenguajes de programación está intrínsecamente ligado a la evolución de las computadoras. A continuación, se presenta una breve reseña sobre su progreso diacrónico (Epitech, 2020; Universia, 202; Rockcontent, 2020; MCPro, 2020; Alcolea, 2019). El primer lenguaje de programación natural fue el lenguaje máquina, fundamentado en el sistema de numeración binario, que emplea los dígitos 1 y 0 para representar instrucciones y datos, permitiendo la comunicación directa con el hardware.

El funcionamiento de las computadoras programables originales se sustentaba en el uso de tarjetas perforadas, donde cada perforación representaba un 1 o 0, según el sistema de codificación. La información se organizaba en función de una estructura determinada que facilitaba la adecuada interpretación de las instrucciones que debían ejecutarse. Empero, el

lenguaje máquina resultaba particularmente complejo, ya que exigía una precisión absoluta: un solo error en la perforación podía desencadenar una ejecución impredecible.

A comienzos de la década de 1950 surgió el lenguaje ensamblador, como una solución al obstáculo que implicaba interactuar con el lenguaje máquina. Dicha alternativa resultó viable, puesto que emplea un conjunto de abreviaturas mnemónicas y una sintaxis específica, fácil de interpretar por el programador. Aun así, este lenguaje no es estándar, dado que cada fabricante de microcontroladores o microprocesadores cuenta con su propia versión. Además de lo anterior, su aplicación requiere un conocimiento minucioso y particular experticia en la arquitectura interna del dispositivo que se desea programar.

Durante la época referida aparecieron los lenguajes de alto nivel, cuya estructura gramatical se aproximaba al lenguaje propio del ser humano. En este contexto emergió la figura del compilador, encargado de traducir el lenguaje de alto nivel; en primera instancia a lenguaje ensamblador y posteriormente a lenguaje máquina. Algunos de los lenguajes listados arriba, en la Tabla 2.1, son ejemplos de alto nivel.

2.1.1. PROGRAMACIÓN CON LENGUAJES NO ESTRUCTURADOS

Entre los lenguajes no estructurados destacan Fortran y BASIC, los cuales carecen de un armazón definido. Como efecto, la complejidad de la secuencia de programación aumenta de la mano con la cantidad de código. Una estrategia para aportar estructura al código es la numeración de sus líneas; de manera que un recurso valioso es el comando `GOTO`, capaz de trasladar el flujo a una línea de código específica para manipular la secuencia de ejecución del programa.

2.1.2. PROGRAMACIÓN CON LENGUAJES ESTRUCTURADOS

En esencia, los lenguajes estructurados surgieron para simplificar el desarrollo de programas complejos. Un aspecto clave de estos lenguajes es que pueden prescindir del comando `GOTO` y sustituirlo por subrutinas y funciones independientes capaces de operar variables locales. Asimismo, se introdujo la característica de recursividad, es decir que una función o subrutina puede apelar a sí misma sin interrumpir la secuencia del programa. En esa misma línea destacan lenguajes estructurados como C, Pascal y Algol.

2.1.3. PROGRAMACIÓN ORIENTADA A OBJETOS

Pese a la eficacia de la programación estructurada en situaciones elementales, cuando se destina a problemas complejos usualmente arroja resultados poco satisfactorios. Las limitaciones que muestra el modelo tradicional de desarrollo de sistemas de información se observan en los siguientes aspectos:

- La complejidad inherente del sistema, que obstaculiza la realización de un mantenimiento completo y dinámico.
- Las discrepancias entre la necesidad del usuario y la funcionalidad del sistema, a menudo a causa de una comunicación ineficiente entre usuario y diseñador.
- La dificultad de llevar a cabo modificaciones cuando hay cambios en el entorno o los requerimientos iniciales; por ejemplo, cuando es necesario incorporar el tratamiento de nuevos tipos de datos, como imágenes, sonido y otros recursos multimedia.

Como respuesta a estas limitaciones, se propuso la programación orientada a objetos (POO), cuyo propósito fue superar los obstáculos del modelo estructurado. Por lo tanto, la POO retoma los principios más eficaces de la programación estructurada y los complementa con conceptos novedosos que potencian la organización de los programas. Entre los lenguajes orientados a objetos predominan C++, Java y C#.

2.2. REGISTROS

Tanto los microcontroladores como los microprocesadores pertenecen a la clasificación de dispositivos digitales consignados a resolver tareas vinculadas al hardware; sin embargo, resulta imprescindible comprender la diferencia que distingue a un elemento del otro. Podría decirse que, en términos generales, los microprocesadores se suelen emplear en computadoras personales, portátiles y sistemas embebidos como Raspberry Pi, Tinker Board, Orange Pi y ODROID, los cuales requieren procesar ingentes volúmenes de información (aunque, como cabe esperar, su uso no se limita a estos rubros). Asimismo, estos dispositivos pueden gestionar una gran cantidad de memoria y múltiples puertos de comunicación, ya que estos últimos operan como sistemas periféricos, es decir, son externos al dispositivo. La Figura 2.1 ilustra dicho concepto.

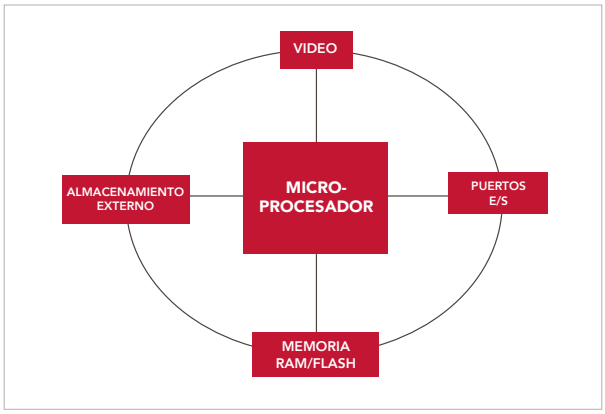


Figura 2.1. Partes de un microprocesador.

En contraste, los microcontroladores predominan en aplicaciones de control de alta especificidad. Como se mencionó, también son categorizados como sistemas embebidos, puesto que el circuito integrado incorpora los componentes necesarios para su funcionamiento. Se debe resaltar que los recursos de un microcontrolador, en lo que respecta a la memoria, son limitados en comparación con un microprocesador (Figura 2.2).

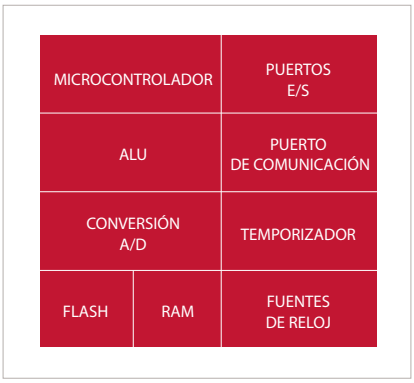


Figura 2.2. Partes de un microcontrolador.

Tanto los microcontroladores como los microprocesadores operan en lenguaje binario, en el cual cada unidad de información se denomina *bit*. Asimismo, los *bits* se agrupan en conjuntos de octetos, conocidos como *bytes*. Ambas clasificaciones de medida son comunes a los dos tipos de dispositivos. El tipo de variable a operar en un programa —bien sea en una computadora personal o un microcontrolador— determina el tamaño que ocupará en *bytes*

un dato puntual en la memoria. En otras palabras, el espacio de la memoria se mide en *bytes* y cada tipo de variable ocupa una cantidad específica. En cuanto al tratamiento de la información, ambos sistemas emplean registros, los cuales fungen como elementos de memoria encargados de almacenar los datos. Por ende, el número de registros requeridos dependerá del tipo de variable utilizada (Figura 2.3). Cada *bit* dentro de un registro se identifica a través de un número, y es conveniente recordar que cada *bit* solo puede adoptar el valor de 0 y 1. En la *praxis*, los datos se representan mediante un conjunto de símbolos denominados *variables*, fundamentales para almacenar un gran rango de valores. En adelante, se hará referencia a estos valores como *datos*.

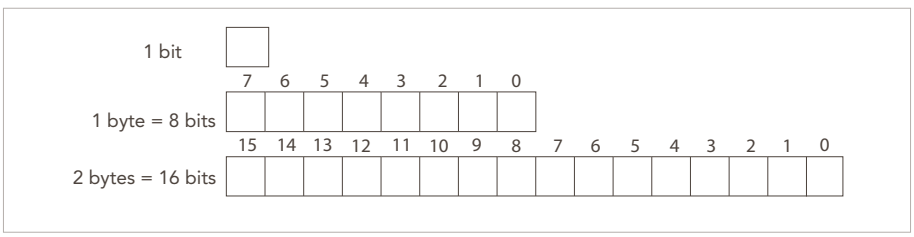


Figura 2.3. Representación de registros, *bytes* y *bits*.

Una de las diferencias angulares entre el lenguaje humano y los sistemas digitales radica en la capacidad de representación. En el primero, se puede referir con total naturalidad a valores inmensos —como el infinito—, o ínfimos —como la longitud de una partícula subatómica—, en teoría, el lenguaje humano es flexible e ilimitado. En contrapartida, los sistemas digitales tienen restricciones, ya que su capacidad de almacenamiento y procesamiento depende del número de *bits*. A raíz de ello, al programar, es necesario definir tipos de variables según cuántos *bits* se emplearán para representar cada dato.

2.3. TIPOS DE DATOS, VARIABLES Y OPERADORES

En la actualidad, el sistema numérico decimal es connatural en la vida cotidiana para llevar a cabo tareas rutinarias. Empero, los sistemas digitales (microcontroladores y microprocesadores) operan con diferentes bases numéricas, entre las que destacan la binaria, la octal y la hexadecimal. A causa de ello, es necesario llevar a cabo conversiones que permitan adaptar y procesar la información de acuerdo con las exigencias de estos sistemas. A manera de

ejemplo, la Tabla 2.2 exhibe la relación entre las bases decimal, binaria, octal y hexadecimal. Es conveniente para el alumno estudiarlas a detalle. Para imprimir los datos se puede usar el Código 2.1.

Tabla 2.2. Relación de datos en binario, octal y hexadecimal.

DECIMAL	BINARIO	OCTAL		HEXADECIMAL	
0	0	0	000 000	0	0000 0000
1	1	1	000 001	1	0000 0001
2	10	2	000 010	2	0000 0010
3	11	3	000 011	4	0000 0011
4	100	4	000 100	4	0000 0100
5	101	5	000 101	5	0000 0101
6	110	6	000 110	6	0000 0110
7	111	7	000 111	7	0000 0111
8	1000	10	001 000	8	0000 1000
9	1001	11	001 001	9	0000 1001
10	1010	12	001 010	A	0000 1010
11	1011	13	001 011	B	0000 1011
12	1100	14	001 100	C	0000 1100
13	1101	15	001 101	D	0000 1101
14	1110	16	001 110	E	0000 1110
15	1111	17	001 111	F	0000 1111
16	10000	20	010 000	10	0001 0000

Código 2.1. Impresión de la tabla de correspondencia de bases numéricas.

```
#include <iostream>
#include <bitset>

using namespace std;

void main()
{
   int i;
   bitset<5> b;
   bitset<6> o;
   bitset<8> h;
```

```
    cout << "Decimal\tBinario\tOctal\t\tHexadecimal" << endl;
    for (i = 0; i <= 16; i++)
    {
         b = i;
         o = i;
         h = i;
         cout << i << '\t' << b;
         cout << '\t' << oct << i << '\t' << o;
         cout << '\t' << hex << i << '\t' << h << endl;
    }
}
```

En la Tabla 2.3 se exponen los mismos valores, para el caso, agrupados en *bytes*. Puede observarse que el valor 4388 requiere 2 *bytes* para representarse.

Tabla 2.3. Representación de datos en tres bases numéricas agrupados en *bytes*.

DECIMAL	BINARIO	HEXADECIMAL
27	00011011	$1B
41	00101001	$29
10	00001010	$0A
93	01011101	$5C
4388	00010001 00100100	$1124

Como se indicó previamente, todo lenguaje de programación requiere del uso de variables para el procesamiento de datos. En el caso de los sistemas embebidos, seleccionarlas correctamente es primordial, puesto que dependerá de ello la gestión de la memoria. Por ejemplo, si se asignan 16 *bits* a una que solo requiere un máximo de 4 *bits*, se generará un desperdicio de espacio. Ahora bien, a diferencia de los sistemas embebidos, una computadora personal cuenta con una capacidad de almacenamiento extenso; por tanto, las pérdidas suelen pasar desapercibidas, excepto cuando los procesos se repiten durante numerosos ciclos. En dicho caso, el rendimiento del sistema puede verse afectado al disminuir la velocidad de procesamiento. Las Tablas 2.4 a 2.7 presentan algunos tipos de variables y sus tamaños en *bytes*.

Tabla 2.4. Representación de datos en Windows 10.

COMPUTADORA PERSONAL (WINDOWS 10) VISUAL STUDIO COMMUNITY 2019 (C++)		
Tipo	*#Bytes*	Valor
`bool`	1	True/False o 1/0
`unsigned char`	1	0 a 255
`unsigned short`	2	0 a 65535
`unsigned int / long`	4	0 a 4294967295
`char`	1	-128 a 127
`short`	2	-32768 a 32767
`int / long`	4	-2147483648 a 2147483647
`float`	4	1.2×10^{-308} a 3.4×10^{308}
`double`	8	2.2×10^{-308} a 3.4×10^{308}

El Código 2.2 muestra un programa en C diseñado para calcular el tamaño en *bytes* de diferentes tipos de variables. La función **`sizeof(type)`** retorna el número de *bytes* requerido por la variable **`type`**, ya sea un valor entero (**`int`**), uno flotante (**`float`**), uno doble (**`double`**), o incluso una estructura o clase.

Tabla 2.5. Tipos de datos, número de *bytes* y rango de valores para PICC Compiler.

MICROCONTROLADOR (PICC COMPILER)		
Tipo	*#Bytes*	Valor
`int1`	1	0 o 1
`char / int8`	1	0 a 255
`int16`	2	0 a 65 535
`int32`	4	0 a 4 294 967 295
`signed int8`	1	-128 a 127
`signed int16`	2	-32 768 a 32 767
`signed int32`	4	-2 147 483 648 a 2 147 483 647
`float`	4	$\pm1.175 \times 10^{-38}$ a $\pm3.402 \times 10^{38}$

Tabla 2.6. Tipos de datos, número de *bytes* y rango de valores para Microchip C18.

MICROCHIP C18 (MANUAL DEL COMPILADOR MICROCHIP C18)		
Tipo	*#Bytes*	Valor
`char / signed char`	1	-128 a 127
`unsigned char`	1	0 a 255
`int / short`	2	-32 768 a 32 767
`unsigned int / unsigned short`	2	0 a 65 535
`short long`	3	-8 388 608 a 8 388 607
`unsigned short long`	3	0 a 16 777 215
`long`	4	-2 147 483 648 a 2 147 483 647
`unsigned long`	4	0 a 4 294 967 295
`float o double`	4	$1.17549435 \times 10^{-38}$ a $6.80564693 \times 10^{38}$

Tabla 2.7. Tipos de datos, número de *bytes* y rango de valores que puede tener Atmel.

ATMEL STUDIO		
Tipo	*#Bytes*	Valor
`bit`	1 *bit*	0/1
`char / signed char`	1	-128 a 127
`unsigned char`	1	0 a 255
`int / shor int / signed int`	2	-32 768 a 32 767
`unsigned int`	2	0 a 65 535
`long int / signed long int`	4	-2 147 483 648 a 2 147 483 647
`unsigned long int`	4	0 a 4 294 967 295
`float o double`	4	$\pm 1.175 \times 10^{-38}$ a $\pm 3.402 \times 10^{38}$

Código 2.2. Obtener el tamaño en *bytes* de diferentes tipos de variables.

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

using namespace std;

void main()
{
   cout << "bool           " << sizeof(bool) << endl;
   cout << "unsigned char  " << sizeof(unsigned char) << endl;
   cout << "unsigned short " << sizeof(unsigned short) << endl;
   cout << "unsigned int   " << sizeof(unsigned int) << endl;
   cout << "char           " << sizeof(char) << endl;
   cout << "short          " << sizeof(short) << endl;
   cout << "int            " << sizeof(int) << endl;
   cout << "float          " << sizeof(float) << endl;
   cout << "double         " << sizeof(double) << endl;
   _getch();
}
```

En esencia, las variables tipo **`bool`** (booleano) pueden adoptar únicamente dos valores: al establecer un valor 0, el estado de la variable será falso (**`false`**); cualquier otro, incluso los negativos, será interpretado como verdadero (**`true`**). Cada variable tiene un rango determinado de valores posibles. Sin embargo, es fundamental considerar qué sucede cuando una variable en su valor máximo incrementa en 1, o bien está en su punto mínimo y decrementa en 1. El Código 2.3 ilustra el comportamiento de los distintos tipos de datos enteros en su punto máximo al incrementar una unidad.

Código 2.3. Valores máximo y mínimo de tipos de datos enteros.

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

using namespace std;

void main()
{
   unsigned char uc1, uc2;
   unsigned short us1, us2;
   unsigned int ui1, ui2;
   char c1, c2;
   short s1, s2;
   int i1, i2;
```

```
    //unsigned char - 1 byte - 8 bit
    uc1 = 255; uc2 = uc1 + 1;
    printf("unsigned char: %u a %u\n", uc1, uc2);
    //char - 1 byte - 8 bit
    c1 = 127; c2 = c1 + 1;
    printf("         char: %d a %d\n", c1, c2);
    //unsigned short - 2 byte - 16 bit
    us1 = 65535; us2 = us1 + 1;
    printf("unsigned short: %u a %u\n", us1, us2);
    //short - 2 byte - 16 bit
    s1 = 32767; s2 = s1 + 1;
    printf("         short: %d a %d\n", s1, s2);
    //unsigned int - 4 byte - 32 bit
    ui1 = 4294967295; ui2 = ui1 + 1;
    printf(" unsigned int: %u a %u\n", ui1, ui2);
    //unsigned int - 4 byte - 32 bit
    i1 = 2147483647; i2 = i1 + 1;
    printf("          int: %d a %d\n", i1, i2);
    _getch();
}
```

La salida correspondiente al Código 2.3 es:

```
unsigned char: 255 a 0
         char: 127 a -128
unsigned short: 65535 a 0
         short: 32767 a -32768
unsigned int: 4294967295 a 0
          int: 2147483647 a -2147483648
```

El Código 2.4 muestra el caso opuesto: cuando una variable en su valor mínimo recibe una resta de 1. Como consecuencia de la operación, su valor se desplazará al máximo representable para ese tipo de variable.

Código 2.4. Valores mínimo y máximo de tipos de datos enteros.

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

using namespace std;
```

```
void main()
{
        unsigned char uc1, uc2;
        unsigned short us1, us2;
        unsigned int ui1, ui2;
        char c1, c2;
        short s1, s2;
        int i1, i2;

        //unsigned char - 1 byte - 8 bit
        uc1 = 0; uc2 = uc1 - 1;
        printf("unsigned char: %u a %u\n", uc1, uc2);
        //char - 1 byte - 8 bit
        c1 = -128; c2 = c1 - 1;
        printf("         char: %d a %d\n", c1, c2);
        //unsigned short - 2 byte - 16 bit
        us1 = 0; us2 = us1 - 1;
        printf("unsigned short: %u a %u\n", us1, us2);
        //short - 2 byte - 16 bit
        s1 = -32768; s2 = s1 - 1;
        printf("         short: %d a %d\n", s1, s2);
        //unsigned int - 4 byte - 32 bit
        ui1 = 0; ui2 = ui1 - 1;
        printf(" unsigned int: %u a %u\n", ui1, ui2);
        //unsigned int - 4 byte - 32 bit
        i1 = -2147483648; i2 = i1 - 1;
        printf("          int: %d a %d\n", i1, i2);
        _getch();
}
```

La salida del Código 2.4 es la siguiente:

```
unsigned char: 0 a 255
         char: -128 a 127
unsigned short: 0 a 65535
         short: -32768 a 32767
 unsigned int: 0 a 4294967295
          int: -2147483648 a 2147483647
```

2.3.1. DECLARACIÓN Y ASIGNACIÓN DE VARIABLES

La declaración de variables consiste en un proceso elemental, pues basta con especificar su tipo y su nombre. Aun cuando la nomenclatura no está normalizada, existen ciertas convenciones informales para aplicarla: algunos programadores optan por separar las palabras

con el carácter de guion bajo (_), mientras que otros se inclinan por usar mayúsculas para diferenciarlas. Por ejemplo:

```
int NumeroCasa;
int numero_casa;
int Numero_Casa
```

Con el propósito de garantizar la comprensión de un código al revisitarlo después de un extenso periodo de inactividad, resulta preciso adoptar el empleo de un sistema convencional de nomenclatura para los programas o bien ceñirse a las reglas previamente definidas por una metodología estandarizada.

2.3.2. ASIGNAR UN VALOR

Es posible asignar un valor a una variable al momento de declararla:

```
int NumeroCasa = 321;
```

Asimismo, es factible declarar múltiples variables en una misma línea; en este caso, todas compartirán el mismo tipo de dato.

```
int I = 0, j = 12;
```

2.3.3. VALORES CON SIGNO Y SIN SIGNO

Una vez que el alumno comprenda que las variables pueden almacenar datos con distintos rangos de valores, podrá avanzar al siguiente nivel. En esta fase de asimilación, es imprescindible para el pupilo enfocarse en la forma como se interpretan los datos, según se trabaje con o sin signo. Por una parte, en aplicaciones donde solo se consideran cantidades positivas, el valor de los datos se obtiene a partir de la conversión directa de binario a decimal. Por el contrario, si se considera también negativos, el *bit* localizado en el extremo izquierdo —*bit* más significativo (*most significant bit*, MSB, por sus siglas en inglés)— se interpreta como el indicador de signo negativo (Figura 2.4).

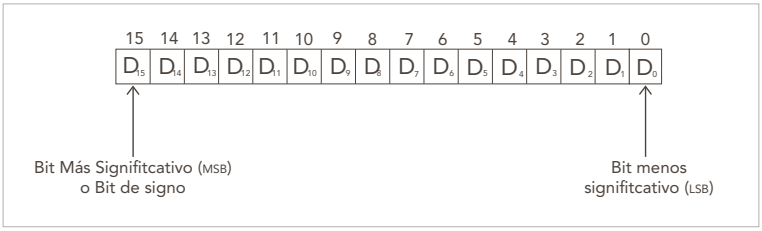


Figura 2.4. Representación de un registro D con MSB y LSB.

Ahora bien, si el MSB equivale a 0, el dato es positivo; de forma opuesta, si es 1 se interpreta negativo. La Figura 2.5 manifiesta la interpretación de datos de 3 *bits* tanto con signo como sin signo. A su vez, la Tabla 2.8 complementa la información de la figura.

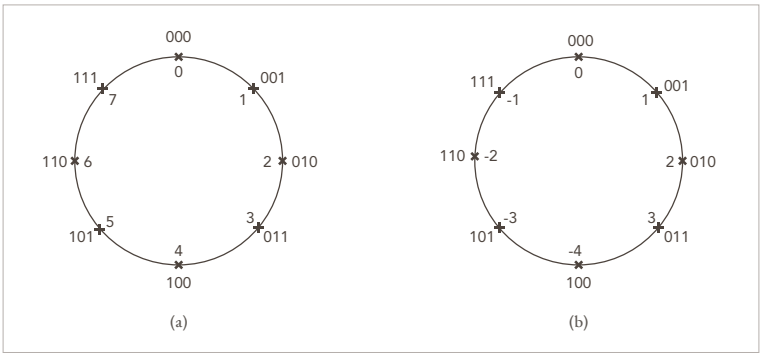


Figura 2.5. Interpretación de datos. (a) 3 *bits* sin signo. (b) 3 *bits* con signo.

Tabla 2.8. Representación de datos de 3 *bits* con signo.

DATO	VALOR REPRESENTADO	
	Con signo	Sin signo
000	0	0
001	1	1
010	2	2
011	3	3
100	-4	4
101	-3	5
110	-2	6
111	-1	7

Las Figuras 2.6 y 2.7 muestran el alcance de valores para datos de 4 y 5 *bits*, tanto con signo como sin signo, haciendo evidente que el valor digital en sí mismo permanece constante. Sin embargo, es posible que el significado del dato varíe según el formato empleado, en otras palabras, si se interpreta como positivo o negativo. Enseguida, se describe el método de representación de valores con punto flotante; es decir, de tipo `float` y `double`.

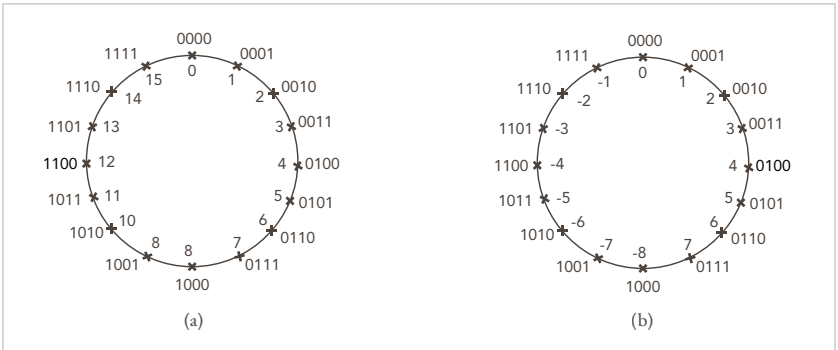


Figura 2.6. Interpretación de datos. (a) 4 *bits* sin signo. (b) 4 *bits* con signo.

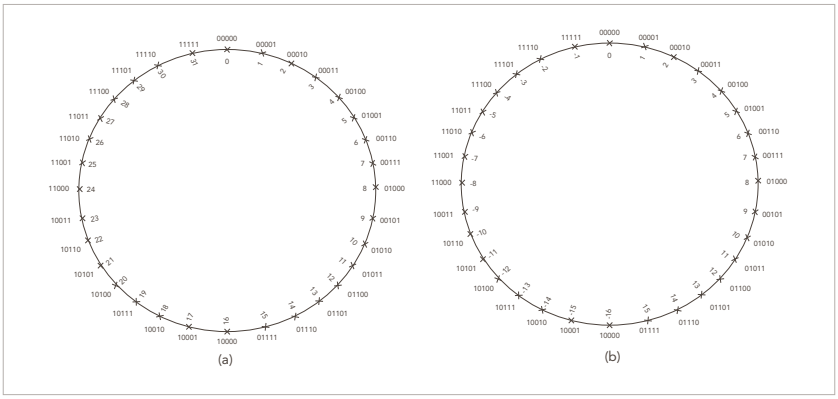


Figura 2.7. Interpretación de datos. (a) 5 *bits* sin signo. (b) 5 *bits* con signo.

2.3.4. VARIABLES TIPO FLOTANTE DE PRECISIÓN SENCILLA IEEE 754

Una variable de tipo flotante se conforma de dos partes: la mantisa y el exponente. La Figura 2.8 apuntala un valor de tipo flotante junto con las partes mencionadas. De ahí que manejar

este tipo de variables implique un elevado nivel de complejidad en los sistemas digitales, ya que su formato consume mayor tiempo de cómputo para realizar operaciones aritméticas.

Figura 2.8. Representación de un valor tipo flotante (`float`).

El formato de una variable de tipo flotante de 32 *bits* está normado por el estándar IEEE 754, según el cual una variable de este tipo se compone de los siguientes *bits*:

- 1 *bit* de signo
- 8 *bits* del exponente
- 23 *bits* del valor representado o fracción (mantisa)

En la Figura 2.9 se observa la forma que guarda un registro de una variable tipo flotante de 32 *bits* (4 *bytes*).

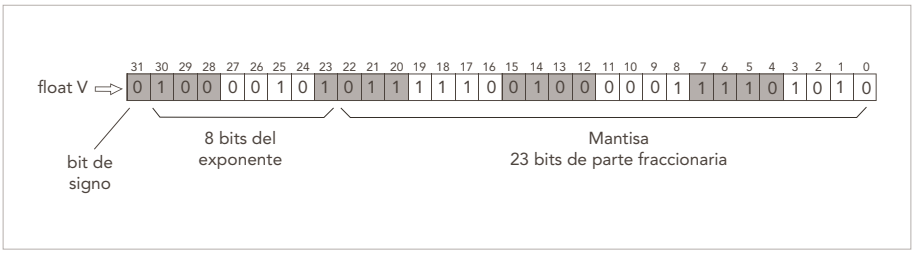


Figura 2.9. Formato de un registro de una variable flotante de 32 *bits*.

Para determinar el valor decimal equivalente con punto flotante (Figura 2.9), se requiere seguir los pasos que se indican a continuación:

El valor decimal de la variable V (Figura 2.9) se calcula conforme a (1). El *bit* más significativo V_{31} indica la polaridad (positiva o negativa), los *bits* V_{22} a V_0 representan el dato correspondiente. Los *bits* V_{30} a V_{23} representan el exponente.

$$V = (-1)^{V_{31}} \times \left(1 + \sum_{i=1}^{23} V_{23-i} 2^{-i}\right) \times 2^{exp-127} \tag{1}$$

En términos generales, los valores del registro se emplean según se muestra en la Figura 2.10.

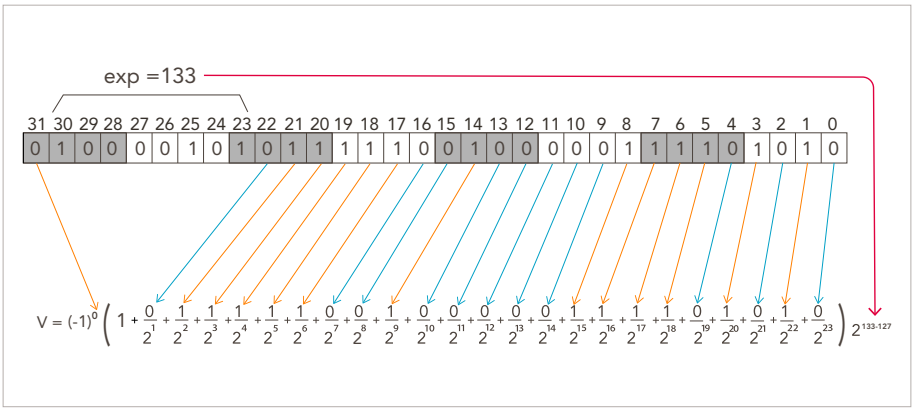


Figura 2.10. Aplicación de la ecuación (1) para obtener el valor de *V*.

El valor de la variable *V* es:

$$V = (-1)^0 \times (1 + 0.486386538) \times 2^{133-127}$$

$$V = 95.128738432$$

Ahora, para realizar el proceso inverso, se convertirá el valor 95.12874 a su representación binaria en formato de punto flotante de precisión simple (32 *bits*), siguiendo el algoritmo que se indica:

Se convierte el valor a binario en al menos 32 *bits*:

$$V = 95.12874_{10} = 1011111.00100000111101010001101011001001 1_2$$

Se normaliza el valor:

$$V = 1.01111100100000111101010001101011001001 1_2 \times 2^6$$

P es el valor de la potencia de 2, por lo tanto, $P = 6$.

Se calcula el exponente en exceso de $2^2 - 1$ por medio de la expresión (2).

$$Exponente = P + (2^{n-1} - 1) \quad (2)$$

$$Exponente = 6_{10} + (2^{8-1} - 1)_{10} = 133_{10}$$

$$Exponente = 10000101_2$$

A continuación, se construye la mantisa tomando los 23 *bits* más significativos a partir del punto decimal. Es decir, se omite el 1 a la izquierda del punto decimal.

$$Mantisa = 01111100100000111101010_2$$

Dado que el valor es positivo, el *bit* correspondiente al signo permanece en cero.

$$Signo = 0$$

Al final, se obtiene la representación binaria de 95.12874 en formato de precisión sencilla.

$$V = 95.12874_{10} = 01000010101111100100000111101010_2 = 42BE41EA_{16}$$

El Código 2.5 verifica la representación binaria del valor 95.12874 en formato de precisión sencilla. El resultado se muestra en notación hexadecimal.

Código 2.5. Representación en binario de un valor flotante.

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

void main()
{
   float j;
   unsigned int k;

   //Representación del valor 95.12874 en 4 bytes
   j = 95.12874;
   memcpy(&k, &j, 4);
```

```
    printf("%f = 0x%X\n", j, k);
    //Ahora, se realiza el proceso contrario
    memcpy(&j, &k, 4);
    printf("0x%X = %f\n", k, j);
    _getch();
}
```

El resultado de la ejecución del Código 2.5 se muestra a continuación:

```
95.128738 = 0x42BE41EA
0x42BE41EA = 95.128738
```

Para un valor negativo se obtiene:

```
-95.128738 = 0xC2BE41EA
0xC2BE41EA = -95.128738
```

2.3.5. VARIABLES TIPO FLOTANTE DE DOBLE PRECISIÓN IEEE 754

Este tipo de variables se compone por 64 *bits*: 1 *bit* para el signo, 11 *bits* para el exponente y 52 *bits* para la mantisa, como se aprecia en la Figura 2.11.

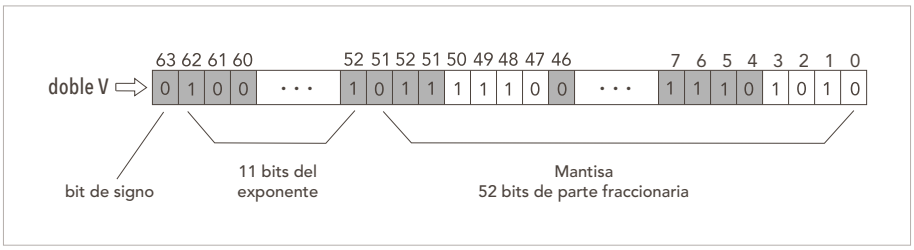


Figura 2.11. Formato binario de un valor flotante de doble precisión.

El procedimiento de conversión es similar al de precisión sencilla, con algunas modificaciones. Para ilustrarlo, se convertirá el valor 95.12874 a su representación binaria en formato de punto flotante de doble precisión (64 *bits*), siguiendo los pasos indicados a continuación.

Se convierte el valor a binario con al menos 64 *bits*:

$$V = 95.12874_{10} = \text{BE41EA35935FC000}_{16}$$

$$V = 1011111.00100000111101010001101011001001101011111110000000000000000_{2}$$

Se normaliza el valor de manera que el punto decimal quede ubicado antes del *bit* más significativo. En otras palabras, el punto decimal se desplaza seis posiciones hacia la izquierda; por lo tanto, de acuerdo a la Ecuación (2), se obtiene $P = 6$.

$$V = 1.01111100100000111101010001101011001001101011111110000_{2} \times 2^{6}$$

$$V = 1.01111100100000111101010001101011001001101011111110000000000000000_{2} \times 2^{6}$$

Se calcula el exponente en exceso de $2^{n-1} - 1$, donde n es el número de *bits* del exponente, por lo que $n = 11$ y $P = 6$:

$$Exponente = P + (2^{n-1} - 1)$$

$$Exponente = P + (2^{n-1} - 1)$$

$$Exponente = 6_{10} + (2^{11-1} - 1)_{10} = 1029_{10}$$

$$Exponente = 10000000101_{2}$$

Ahora, se forma la mantisa seleccionando los 52 *bits* más significativos a partir del punto decimal; es decir, omitiendo el 1 a la izquierda del punto decimal:

$$Mantisa = 0111110010000011110101000110101100100110101111111000_{2}$$

Como el valor es positivo, el *bit* de signo se mantiene en cero:

$$Signo = 0$$

Al final, se forma la representación de la cantidad 95.12874 en binario, con doble precisión:

$$V = 95.12874_{10} = 4057C83D46B26BF8_{16} = 0100000001010111110010000011110101000110101100100110101111111000_2$$

El Código 2.6 permite verificar la representación del valor 95.12874 en binario utilizando el formato de doble precisión. El resultado se presenta en hexadecimal.

Código 2.6. Representación de un valor `double` en binario con formato de doble precisión.

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

using namespace std;

void main()
{
    double j;
    unsigned int k[2];

    //Representación del valor 95.12874 en 8 bytes
    j = 95.12874;
    memcpy(&k, &j, 8);
    printf("%f = 0x%X%X\n", j, k[1],k[0]);
    //Ahora, se realiza el proceso contrario
    memcpy(&j, &k, 8);
    printf("0x%X%X = %f\n", k[1], k[0], j);
    _getch();
}
```

El resultado es el siguiente:

```
95.128740 = 0x4057C83D46B26BF8
0x4057C83D46B26BF8 = 95.128740
```

Para un valor negativo, el resultado es:

```
-95.128740 = 0xC057C83D46B26BF8
0xC057C83D46B26BF8 = -95.128740
```

2.3.6. ERROR AL DEFINIR UN TIPO DE VARIABLE

¿Qué ocurre si el tipo de variable elegido es demasiado restringido? Si la cifra asignada excede el máximo admitido por el tipo de variable, el valor almacenado puede diferir del original. Por ejemplo, si a una variable de 16 *bits* se le asigna el valor 65535_{10}, este se guardará de forma correcta. En cambio, si se asigna el valor 65536_{10}, la cifra registrada será igual a 0. Lo anterior se debe a que el dato 65535_{10} precisa de 16 *bits*, mientras que 65536_{10} requiere de 17 *bits*. Debido a que la variable solamente maneja 16 *bits*, el *bit* en la posición 17 se descarta. Del mismo modo, si se intenta almacenar en una variable de 16 *bits* la cantidad de 78435_{10}, únicamente se guardará el valor 12899_{10}. Este comportamiento se ilustra en la Figura 2.12.

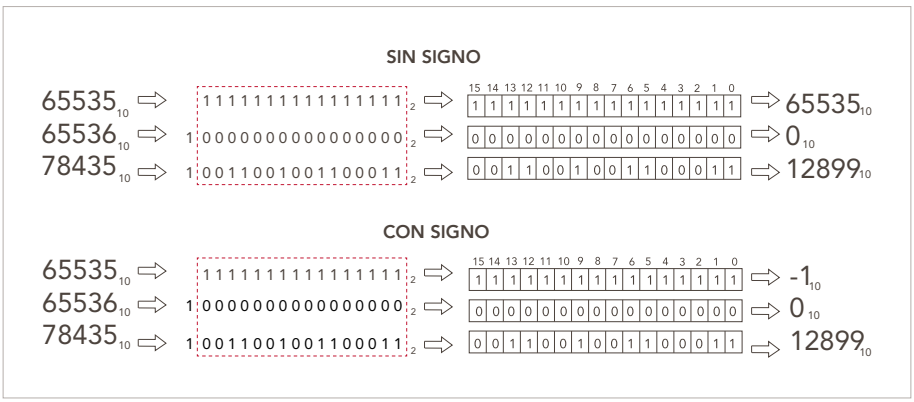


Figura 2.12. Ajuste de valores en variables de 16 *bits*.

El Código 2.7 presenta los resultados de asignar distintos valores a una variable de 16 *bits*, como se muestra en la Figura 2.12. Las variables tipo `short` ocupan 2 *bytes* (16 *bits*).

Código 2.7. Ejemplo de ajuste de valores.

```
#include <iostream>
#include <conio.h>
using namespace std;

void main()
{
   unsigned short A, B, C;
   short D, E, F;
```

```
    A = 65535;
    B = 65536;
    C = 78435;
    cout << "Sin signo:\n";
    cout << "A = " << A << "\tB = " << B << "\tC = " << C << endl;
    D = 65535;
    E = 65536;
    F = 78435;
    cout << "Con signo:\n";
    cout << "D = " << D << "\t\tE = " << E << "\tF = " << F << endl;
    _getch();
}
```

La salida desplegada en pantalla es:

```
Sin signo:
A = 65535       B = 0   C = 12899
Con signo:
D = -1          E = 0   F = 12899
```

2.3.7. VARIABLES LOCALES Y GLOBALES

En C++, al igual que en diversos lenguajes de programación, existen variables locales y globales. Las globales pueden ser utilizadas en cualquier parte del programa, mientras que las locales solo son válidas dentro del bloque de código donde fueron declaradas. En el Código 2.8 se muestra un ejemplo de la variable local `k`, cuyo valor se modifica dentro de la función `prueba()`.

Código 2.8. Uso de una variable global `k`.

```
#include <iostream>

using namespace std;

float k;          //Declaración de variable global

void prueba(void);       //Declaración del prototipo prueba

void main()
```

```
{
   cout << k << endl;    //Se imprime el valor de k
   prueba();  //La variable global es modificada en otra función
   cout << k << endl;    //Se imprim el valor de k
}

void prueba(void)        //Definición de prueba
{
   k = 34.56873;         //Se puede modificar el valor de k
                                 //debido a que es una variable global
}
```

El resultado de ejecutar este código es:

```
0
34.5687
```

El compilador asignó de manera predeterminada el valor inicial de 0 a la variable **k** al momento de su creación. La función **prueba()** modifica este valor y, al retornar al lugar desde donde fue invocada, **k** conserva el valor 34.56873. Por otro lado, si **k** fuera una variable local, el compilador marcaría error, ya que al ser local en **main()**, su valor no es modificable desde otra función. Tal restricción se debe a que las variables locales solo tienen validez dentro del bloque de instrucciones delimitado por {} en el que fueron declaradas. En el Código 2.9 se ejemplifica qué causa este error al compilar, el cual generalmente se manifiesta con el mensaje "identificador no definido".

Código 2.9. Demostración del uso de una variable local.

```
#include <iostream>

using namespace std;

void prueba(void);               //Declaración del prototipo prueba

void main()
{
        float k;          //Declaración de variable local (Se encuentra
                                 //dentro de un bloque de instrucciones)
        cout << k << endl;       //Se imprime el valor de k
        prueba();        //La variable global es modificada en otra función
        cout << k << endl;       //Se imprime el valor de k
}
```

```
void prueba(void)        //Definición de prueba
{
        k = 34.56873;    //No se puede modificar el valor de k
                            //debido a que es una variable local de main()
                            //y la variable k no está definida en prueba()
}
```

Cuando ocurre el caso de que una variable global es homónima a una local, al ejecutar un bloque de instrucciones donde ambas son admitidas, la que prevalece es la local. En consecuencia, cualquier operación realizada sobre la variable afectará únicamente a esta última. De forma paralela, cuando por algún motivo una variable local es destruida, el valor que se le ha asignado se pierde; por ello, tan pronto como se vuelva acceder al bloque de instrucciones, la variable local debe crearse nuevamente e inicializar su valor. Este proceso se muestra en el Código 2.10, donde `k` es local en la función `prueba()`: cada que dicha función es activada, `k` se establece en 0 y posteriormente se incrementa su valor. Al finalizar la ejecución de `prueba()`, la variable `k` es liberada.

Código 2.10. Manejo de una variable local `k`.

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

void prueba(void);       //Declaración del prototipo prueba

void main()
{
   prueba();     //Se ejecuta la función prueba()
   prueba();     //Se ejecuta la función prueba()
   prueba();     //Se ejecuta la función prueba()
}

void prueba(void)        //Definición de prueba
{
   float k=0;     //Declaración de variable local (Se encuentra
                  //dentro de un bloque de instrucciones)
                  //y se inicializa con 0.
   k = k + 0.345;             //Se asigna un nuevo valor a k
   cout << k << endl;    //Se imprime el valor de k
}
```

La ejecución del código da como resultado:

```
0.345
0.345
0.345
```

C++ permite manipular variables estáticas, las cuales se caracterizan por conservar el valor asignado aún después de ser destruidas. En el Código 2.11 se declara la variable `k` como flotante estática. Es decir, al convocar la función `prueba()`, por segunda ocasión y en adelante, el valor consignado a `k` se mantiene, de forma que puede reutilizarse aunque la ejecución de la función haya concluido.

Código 2.11. Uso de variables estáticas.

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

void prueba(void);       //Declaración del prototipo prueba

void main()
{
   prueba();     //Se ejecuta la función prueba()
   prueba();     //Se ejecuta la función prueba()
   prueba();     //Se ejecuta la función prueba()
}

void prueba(void)        //Definición de prueba
{
   static float k=0;     //Declaración de variable estática
                                 //tipo flotante.
   k = k + 0.345;               //Se asigna un nuevo valor a k
   cout << k << endl;    //Se imprime el valor de k
}
```

De este modo se obtiene:

```
0.345
0.69
1.035
```

Lo anterior evidencia que el valor de `k` se conserva incluso después de terminar la ejecución de la función `prueba()`.

2.3.8. OPERADORES

Se trata de símbolos que indican la ejecución de una operación sobre uno o dos elementos denominados "operandos", dentro de una línea de comando. Se distinguen tres categorías en función de cuántos elementos interactúan (Tabla 2.9):

- **Unario**: Requiere de un solo operando. Se utiliza para marcar los incrementos (++) y los decrementos (--).
- **Binario**: Involucra dos operandos. Ejemplos elementales incluyen suma (+), resta (-), multiplicación (*) y división (/).
- **Ternario**: Solicita tres operandos y se emplea con el objetivo de simplificar expresiones condicionales.

Tabla 2.9. Tipos de operadores.

OPERADOR UNARIO		OPERADOR BINARIO		OPERADOR TERNARIO	
A++	El valor de A se incrementa en 1.	A+B	Suma aritmética	X = (A>B)?E:F	X toma el valor de E si A > B; de lo contrario, toma el valor de F.
A--	El valor de A se decrementa en 1.	A-B	Resta aritmética		
++A	El valor de A se incrementa en 1.	A*B	Multiplicación aritmética		
--A	El valor de A se decrementa en 1.	A/B	División aritmética		

Los operadores unarios actúan según su ubicación respecto de la variable. En el Código 2.12, se utiliza el operador unario ++ como sufijo, lo que indica que primero se realiza la operación y después se incrementa el valor. En este caso, el programa imprime `x = 4, y = 3`.

Código 2.12. Operador unario ++ usado como sufijo.

```
#include <iostream>
#include <conio.h>
void main()
{
    int x, y;

    x = 3;
    y = x++;
    printf("x = %d, y = %d\n", x, y);
    _getch();
}
```

El Código 2.13 muestra el uso del operador unario como prefijo. Primero se ejecuta el incremento y luego se realiza la operación. Al final, los valores de las variables son `x = 4, y = 4`.

Código 2.13. Operador unario ++ usado como prefijo.

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

void main()
{
   int x, y;

   x = 3;
   y = ++x;
   printf("x = %d, y = %d\n", x, y);
   _getch();
}
```

De manera alternativa a la categorización por número de argumentos, los operadores pueden ser clasificarse también en: **de asignación**, **aritméticos**, **para apuntadores**, **relacionales** y **lógicos**.

Operadores de asignación

Los operadores de asignación posibilitan almacenar el resultado de una operación directamente en una variable, simplificando la escritura de la línea de comando. Tales operadores se muestran en la Tabla 2.10 junto con su equivalente. Cabe destacar que no todos los lenguajes de programación incorporan dichos operadores.

Tabla 2.10. Operadores de asignación.

OPERADOR	TIPO DE ASIGNACIÓN	EJEMPLO	EQUIVALENCIA
=	Básica	A = 13	A = 13
*=	Multiplicación	A *= 13	A = A * 13
/=	División	A /= 13	A = A / 13
+=	Suma	A += 13	A = A + 13
-=	Resta	A -= 13	A = A - 13
%=	Módulo	A %= 13	A = A % 13
>>=	Corrimiento lógico a la derecha	A >>= 3	A = A >> 3

OPERADOR	TIPO DE ASIGNACIÓN	EJEMPLO	EQUIVALENCIA
<<=	Corrimiento lógico a la izquierda	A <<= 3	A = A << 3
&=	Operación lógica AND *bit* a *bit*	A &= 55	A = A & 55
|=	Operación lógica OR *bit* a *bit*	A |= 55	A = A | 55
^=	Operación lógica XOR *bit* a *bit*	A ^= 55	A = A ^ 55
= !	Operación lógica NOT	A = !B	A = !B

Las operaciones lógicas se ejecutan *bit* a *bit*, lo que implica que solamente pueden aplicarse a datos enteros y que ambos operandos deben ser del mismo tipo. Las tablas de verdad correspondientes a las operaciones lógicas AND, OR, XOR y NOT se presentan en la Tabla 2.11.

Tabla 2.11. Tablas de verdad de los operadores lógicos.

AND (&)		
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR (|)		
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT (!)	
A	Y
0	1
1	0

XOR (^)		
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Operadores aritméticos

Se trata de recursos utilizados para llevar a cabo operaciones matemáticas elementales, como suma, resta, multiplicación y división. Los principales se exhiben en la Tabla 2.12.

Tabla 2.12. Operadores aritméticos.

OPERADOR	+	−	*	/	%	++	−−
OPERACIÓN	Suma	Resta	Multiplicación	División	Módulo	Incremento en 1	Decremento en 1

Operadores para apuntadores

Estos operadores se destinan para manejar la memoria señalada por un apuntador, de forma que permiten efectuar operaciones con direcciones de memoria. Las principales maniobras con apuntadores se recopilan en la Tabla 2.13.

Tabla 2.13. Operadores para apuntadores.

OPERADOR	DESPLAZAMIENTO
+	Ascendente
–	Descendente o distancia entre elementos
++	Ascendente de 1 elemento
––	Descendente de 1 elemento

Operadores relacionales

También llamados de comparación, los operadores relacionales se utilizan para contrastar dos valores. Los principales se muestran en la Tabla 2.14, y en general son empleados en estructuras condicionales, como `if`, o en los ciclos `while()` y otros bucles.

Tabla 2.14. Operadores relacionales.

OPERADOR	CONDICIÓN
==	Igual que
!=	Diferente a
<	Menor que
>	Mayor que
>=	Mayor o igual que
<=	Menor o igual que

Operadores lógicos relacionales

Este tipo de operadores se aprovecha para enlazar dos o más condiciones, particularmente en los ciclos y en la instrucción `if`. Existen tres operaciones lógicas principales: AND (`&&`),

OR (||) y NOT (!). La Tabla 2.15 manifiesta la relación entre estos operadores, donde las entradas A y B representan los resultados individuales de cada condición, y la salida Y refleja el resultado total de la expresión lógica.

Tabla 2.15. Tablas de verdad de los operadores lógicos relacionales.

AND (&&)		
A	B	Y
Falso	Falso	Falso
Falso	Verdadero	Falso
Verdadero	Falso	Falso
Verdadero	Verdadero	Verdadero

OR (||)		
A	B	Y
Falso	Falso	Falso
Falso	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Verdadero	Verdadero	Verdadero

NOT (!)	
A	Y
Falso	Verdadero
Verdadero	Falso

Como se estableció arriba, los operadores lógicos se distinguen por ejecutar operaciones *bit* a *bit*. La Figura 2.13 ilustra el resultado de aplicar estas operaciones a dos datos de 1 *byte*: un valor de 197 sin signo, en comparación con uno de 85. El Código 2.14 ejemplifica las operaciones usando la clase `bitset`, mientras que en el Código 2.15 se usa la variable tipo `char`.

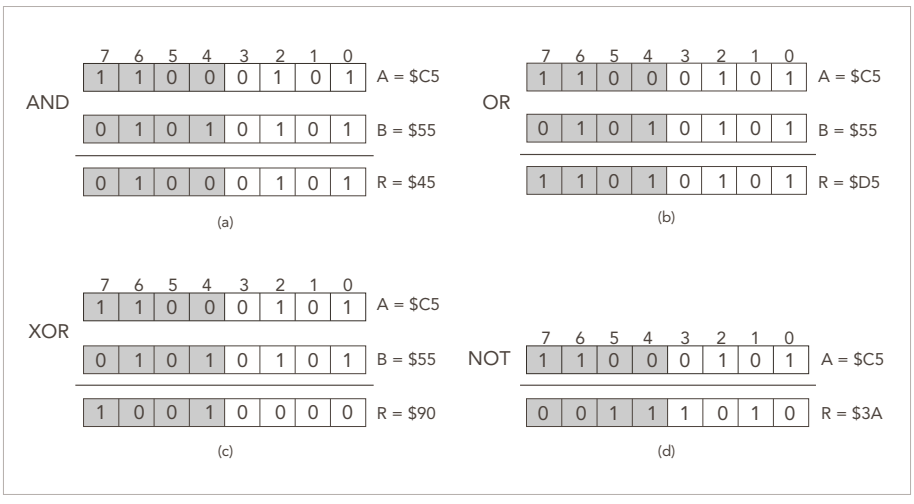


Figura 2.13. Operaciones lógicas. (a) AND. (b) OR. (c) XOR. (d) NOT.

Código 2.14. Operaciones lógicas con `bitset`.

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

void main()
{
   int i;
   bitset<8> A, B, R;

   A = 197; B = 85;
   R = A & B; cout << "A and B = " << R;
   printf(" = %X\n", R);
   R = A | B; cout << "A or  B = " << R;
   printf(" = %X\n", R);
   R = A ^ B; cout << "A xor B = " << R;
   printf(" = %X\n", R);
   R = ~A;    cout << "  not A = " << R;
   printf(" = %X\n", R);
}
```

Código 2.15. Operaciones lógicas usando variables tipo `char`.

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

void main()
{
   int i;
   char A, B, R;

   A = 197; B = 85;
   R = A & B; printf("A and B = %X\n", R);
   R = A | B; printf("A or  B = %X\n", R);
   R = A ^ B; printf("A xor B = %X\n", R);
   R = ~A;    printf("not A   = %X\n", R);
}
```

Las operaciones lógicas de corrimiento consisten en desplazar los *bits* hacia la derecha o la izquierda, según la función aritmética requerida. La Figura 2.14 demuestra este proceso, exhibiendo dos corrimientos a la derecha (de 1 y 4 *bits*) y dos a la izquierda (de 1 y 4 *bits*). Desde la perspectiva binaria, desplazarse un *bit* a la izquierda es equivalente a una multiplicación por 2; de ese modo, un movimiento hacia la derecha equivale a una división entre 2. En la práctica, realizar 3 corrimientos a la derecha implicaría multiplicar la cantidad por 8; puesto

en otras palabras, 2^3, donde el exponente 3 es el número de *bits* a desplazar. En cambio, si se efectúa hacia la izquierda, corresponde dividir la cantidad entre 8.

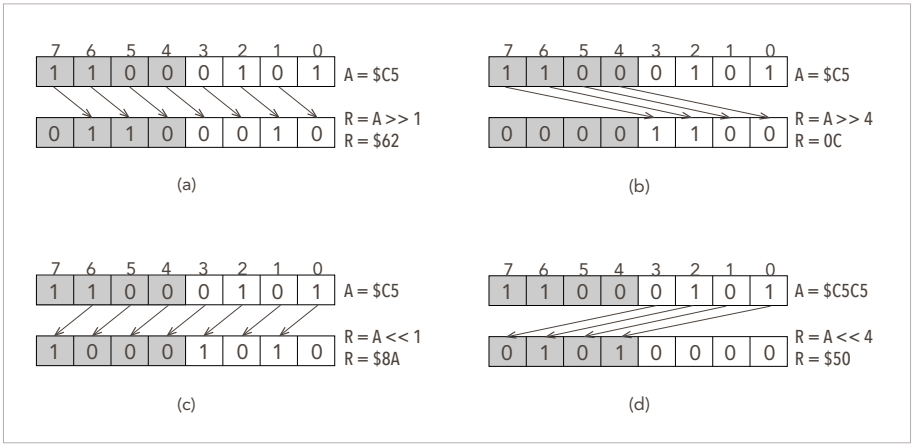


Figura 2.14. Ejemplo de corrimiento lógico para datos de 1 *byte*. (a) De 1 *bit* a la derecha. (b) De 4 *bits* a la derecha. (c) De 1 *bit* a la izquierda. (d) De 4 *bits* a la izquierda.

El Código 2.16 demuestra cómo ejecutar los corrimientos empleando la categoría **`bitset`**. Por otro lado, el Código 2.17 presenta el caso aplicado a variables de tipo **`char`**.

Código 2.16. Aplicación de corrimiento lógico a variables tipo `bitset`.

```
#include <iostream>
#include <bitset>
#include <conio.h>
using namespace std;
void main()
{
    int i;
    bitset<8> A, R;

    A = 197;
    R = A >> 1; cout << A << " >> 1 = " << R << endl;
    R = A >> 4; cout << A << " >> 4 = " << R << endl;
    R = A << 1; cout << A << " << 1 = " << R << endl;
    R = A << 4; cout << A << " << 4 = " << R << endl;
    _getch();
}
```

Código 2.17. Corrimiento lógico a variables tipo char.

```
#include <iostream>
#include <bitset>
#include <conio.h>
using namespace std;
void main()
{
    int i;
    char A, R;

    A = 197;
    R = A >> 1; printf("%X >> 1 = %X\n", A, R);
    R = A >> 4; printf("%X >> 4 = %X\n", A, R);
    R = A << 1; printf("%X << 1 = %X\n", A, R);
    R = A << 4; printf("%X << 4 = %X\n", A, R);
    _getch();
}
```

La Figura 2.15 muestra la aplicación del corrimiento lógico para datos de 16 *bits*.

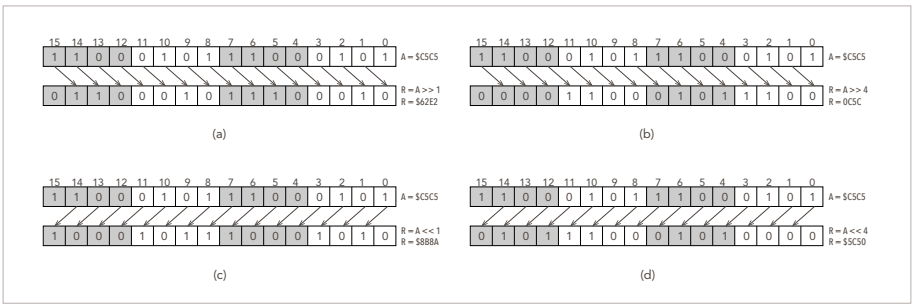


Figura 2.15. Ejemplo de corrimiento lógico para datos de 2 *bytes*.
(a) De 1 *bit* a la derecha. (b) De 4 *bits* a la derecha. (c) De 1 *bit* a la izquierda. (d) De 4 *bits* a la izquierda.

Condicionales

En programación, los condicionales son procedimientos orientados a evaluar situaciones posibles y determinar cursos de acción. Por ejemplo, para decidir si un bloque de instrucciones se ejecuta o no. Estas condiciones se cimientan a partir de los operadores relacionales mostrados en la Tabla 2.14. Ahora bien, la instrucción `if` constituye la forma más elemental de implementar un condicional. Su modo de accionar se basa en la evaluación de una condición: si resulta satisfecha (`true`), se ejecuta el bloque de instrucciones correspondiente:

```
if(condición o prueba lógica)
 Lo que pasa si se cumple la condición
```

También puede llevar a cabo sentencias o acciones cuando no se cumple (**`false`**).

```
if(condición)
 Lo que pasa si se cumple la condición
else
 Lo que pasa si no se cumple la condición
```

La Figura 2.16 expone un diagrama de flujo que ilustra la funcionalidad de la condición **`if`**.

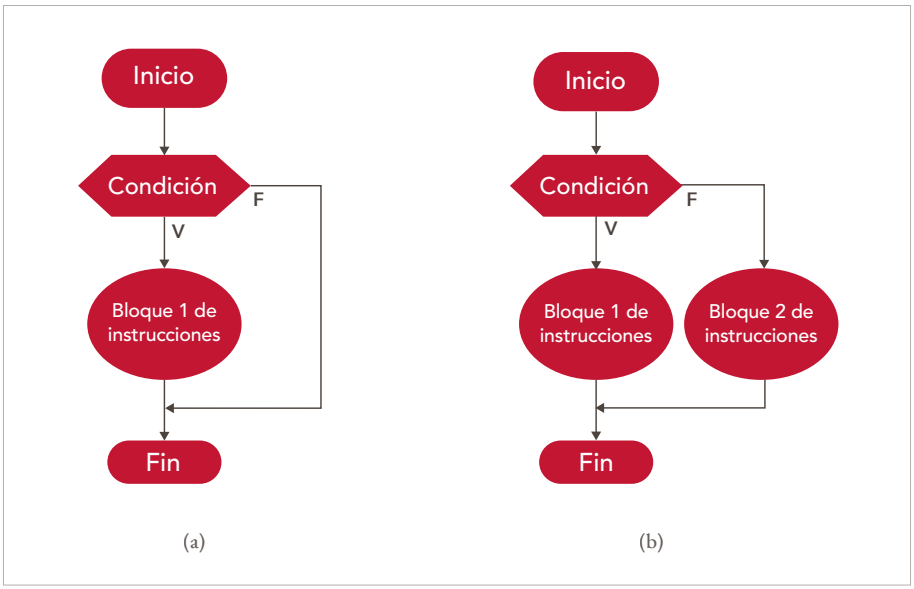


Figura 2.16. Diagrama de flujo de una condición. (a) Solo salida verdadera. (b) Salidas verdadera y falsa.

La estructura condicional **`if/else`** puede interpretarse como "si" y "de lo contrario". Este tipo de expresiones son utilizadas en la vida habitual, porque son inherentes en el proceso de elecciones. Por ejemplo, antes de adquirir un producto, se consulta su costo y se evalúa la capacidad económica personal. Si la capacidad es suficiente, se realiza la compra; de lo contrario, existen dos alternativas: retirarse o buscar una opción más asequible. El diagrama de flujo correspondiente se muestra en la Figura 2.17.

Figura 2.17. Ejemplo de un diagrama de flujo para la toma de decisiones.

El Algoritmo 2.1 muestra el uso del `if` una vez que la condición se cumple; por su parte, el Algoritmo 2.2 discurre ambos escenarios: cuando es satisfecha o insatisfecha. El diagrama de flujo correspondiente a este último caso se presenta en la Figura 2.17.

Algoritmo 2.1. Uso de `if` con solo respuesta verdadera.

```
1. if(Dinero disponible >= costo).
2. Se adquiere el producto.
3. Se retira del establecimiento.
```

Algoritmo 2.2. Funcionamiento de `if` considerando el resultado de la condición satisfecha o insatisfecha.

```
1. if(Dinero disponible >= costo)
2                Se adquiere el producto.
3. Else
4.       Se busca otra opción de compra.
5. Se retira de la tienda.
```

La sintaxis de la condición `if` en lenguaje C, así como en otros lenguajes de programación, se presenta en el Código 2.18.

Código 2.18. Sintaxis de la sentencia `if`.

```
if(condición)
{
    //Bloque de operaciones 1
}
else
{
    //Bloque de operaciones 2
}
```

Cuando el bloque de instrucciones contiene tan solo una línea, las llaves de apertura y cierre pueden omitirse. Además, existe una forma simplificada de utilizar el `if`, empleada para la asignación de un valor a una variable. En el siguiente ejemplo, la variable `X` toma el valor de 3 si la condición `A > B` es falsa, y 2 si la condición es verdadera. Tal estructura resulta útil para asignaciones directas; empero, no sirve para ejecutar múltiples instrucciones, y su sintaxis está sujeta al lenguaje de programación.

```
X = (A>B)?2:3;
```

A continuación, se presentan algunos ejercicios que ejemplifican el uso del condicional `if` en la resolución de problemas básicos.

Ejercicio: Identificar si un número es par o impar

Comprensión del problema

Un número *N* se considera impar si, al dividirlo entre 2, el residuo es igual a 1. Otra forma de identificarlo es cuando la división entre 2 produce una parte decimal 0.5. Por ejemplo:

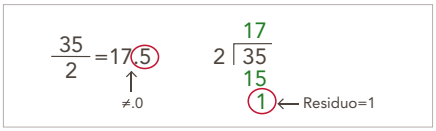


Figura 2.18. Identificar si un número es par o impar.

Algoritmo

A continuación, se describen las etapas a realizar para establecer la solución del ejercicio.

Algoritmo 2.3. Determinación de la paridad de un número.

```
1. Se pide el valor de N.
2. Se calcula el módulo 2 a N.
3. ¿El resultado del módulo es 1? Sí: Ir al paso 4, No: Ir al paso 6.
4. Imprimir “N es impar”.
5. Ir al paso 7.
6. Imprimir “N es par”.
7. Fin.
```

Diagrama de flujo

En la Figura 2.19 se presenta el diagrama de flujo correspondiente, en el cual se identifican los pasos del procedimiento. En el esquema, la flecha orientada hacia la izquierda se utiliza para indicar la operación de asignación. En otras palabras, el valor ubicado en el lado derecho se aplica a la variable situada a la izquierda.

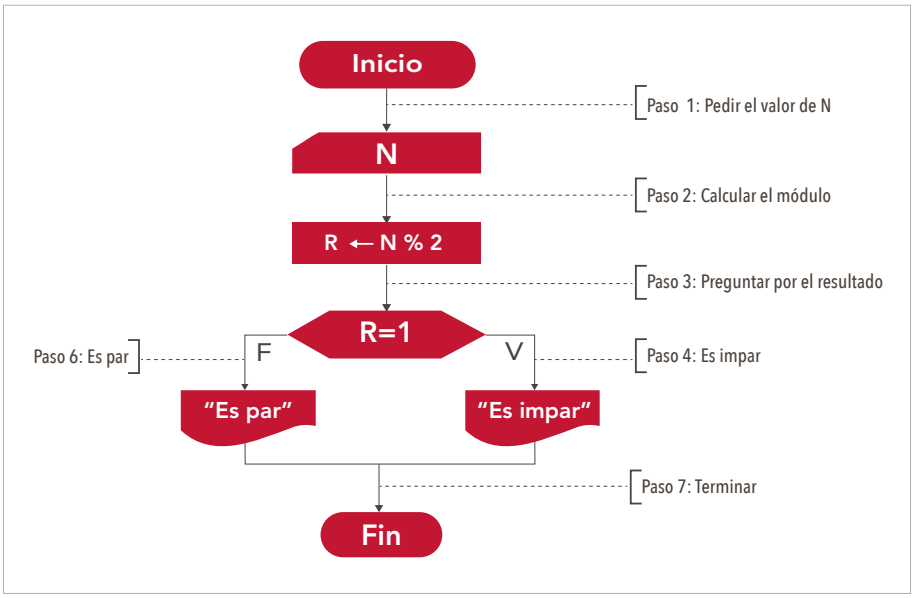


Figura 2.19. Número par o impar.

Código en C

Para la elaboración del programa es imprescindible conocer las características del lenguaje de programación, especialmente los operadores. En este caso, de acuerdo con la Tabla 2.12, se emplea el operador módulo (`%`), el cual devuelve el residuo entero de una división. Por lo tanto, es idóneo para la resolución del problema. Es necesario analizar el código e identificar su relación de correspondencia con el diagrama de flujo mostrado en la Figura 2.19.

Código 2.19. Número par o impar.

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

void main()
{
    int N;
    int R;

    N = 35;                 //Paso 1: Conocer el valor de N
    R = N % 2;     //Paso 2: Calcular el módulo 2
    if (R == 1)             //Paso 3: Preguntar por el resultado
         printf("Es impar");        //Paso 4
    else
         printf("Es par");          //Paso 6
    //Paso 7
    _getch();
}
```

La condición `if(R == 1)` puede simplificarse como `if(R)`. En esta forma, se considerará falsa únicamente cuando `R` equivalga a cero; para cualquier valor distinto de cero, se evaluará como verdadera.

Ejercicio: Calcular las raíces de una ecuación cuadrática

Comprensión del problema

La ecuación cuadrática se expresa en la forma $Ax^2 + Bx + C = 0$. Sus raíces pueden calcularse mediante la fórmula cuadrática:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Existen tres casos de solución para la ecuación cuadrática. El primero se lleva a cabo cuando $B^2 - 4AC > 0$, el valor de las dos raíces es real:

$$x_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \qquad x_2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

Por ejemplo, en la ecuación $x^2 - x - 12 = 0$, los coeficientes son $A = 1$, $B = -1$ y $C = -12$. A partir de ello se obtiene:

$$B^2 - 4AC = (-1)^2 - 4(1)(-12)$$

$$B^2 - 4AC = 49$$

$$\therefore B^2 - 4AC > 0$$

Por lo tanto:

$$x_1 = \frac{-(-1) + \sqrt{49}}{2(1)} = \frac{1+7}{2} \qquad x_1 = \frac{-(-1) - \sqrt{49}}{2(1)} = \frac{1-7}{2}$$

$$x_1 = 4 \qquad x_2 = -3$$

El segundo caso se da cuando $B^2 - 4AC = 0$; ambas raíces coinciden y tienen el mismo valor real, tal como se muestra en la ecuación:

$$x_1 = x_2 = \frac{-B}{2A}$$

Para la ecuación $x^2 - 6x + 9 = 0$, se identifican los coeficientes: $A = 1$, $B = -6$, $C = 9$. En este caso, se obtiene:

$$B^2 - 4AC = (-6)^2 - 4(1)(9) = 0$$

Entonces:

$$x_1 = x_2 = \frac{-(-6)}{2(1)}$$

$$x_1 = x_2 = 3$$

El último caso, cuando $B^2 - 4AC < 0$, ambas raíces corresponden a valores complejos.

$$x_1 = \frac{-B + \sqrt{-(B^2 - 4AC)i}}{2A}$$

$$x_2 = \frac{-B - \sqrt{-(B^2 - 4AC)i}}{2A}$$

Por ejemplo, considerando la ecuación $4x^2 + 12x + 13 = 0$, los coeficientes correspondientes son $A = 4$, $B = 12$, $C = 13$. En este caso, se tiene que:

$$B^2 - 4AC = (12)^2 - 4(4)(13) = 0$$

$$B^2 - 4AC = -64$$

Por tanto:

$$x_1 = \frac{-12 + \sqrt{-(-64)i}}{2(4)} = \frac{-12 + 8i}{8}$$

$$x_1 = -\frac{3}{2} + i$$

$$x_2 = \frac{-12 - \sqrt{-(-64)i}}{2(4)} = \frac{-12 - 8i}{8}$$

$$x_2 = -\frac{3}{2} - i$$

Algoritmo

El Algoritmo 2.4 considera todos los casos a resolver.

Algoritmo 2.4. Solución de una ecuación cuadrática.

```
1. Pedir los valores de A, B y C.
2. Calcular el valor del radical.
3. ¿El radical es menor que 0? Sí: Ir al paso 4. No: Ir al paso 7.
4. Calcular las dos raíces: complejas.
5. Imprimir el valor de las dos raíces.
6. Ir al paso 9.
7. Calcular las dos raíces reales.
8. Imprimir el valor de las dos raíces.
9. Fin.
```

Diagrama de flujo

El cálculo de la raíz cuadrada se lleva a cabo mediante la función `sqr()`. Cabe señalar que la función puede variar según el lenguaje empleado.

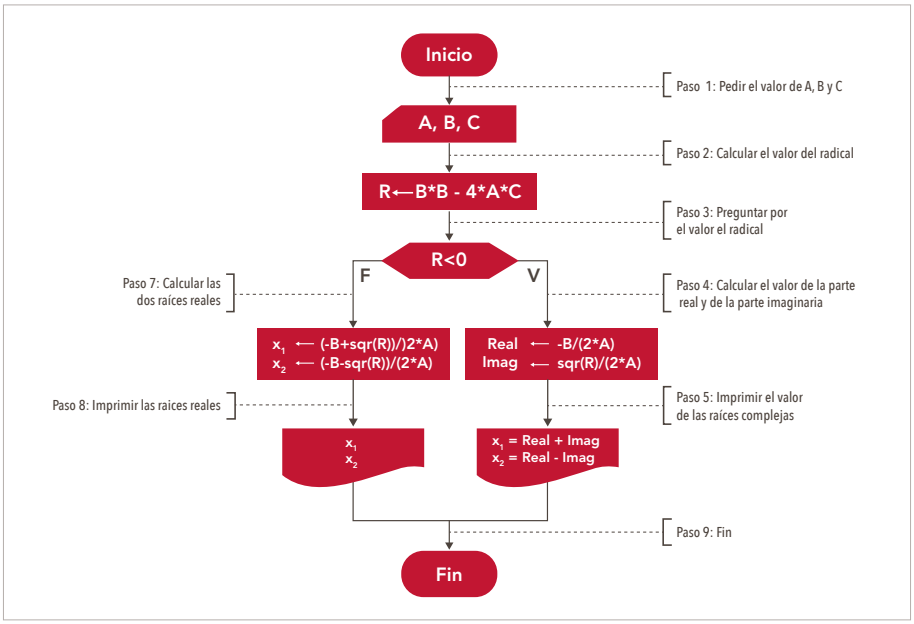


Figura 2.20. Solución de la ecuación cuadrática.

Código en C

Las líneas que conforman el programa para resolver el problema se muestran a continuación en el Código 2.20.

Código 2.20. Solución de una ecuación cuadrática.

```
#include <iostream>
#include <math.h>
using namespace std;

void main()
{
    //Paso 0: Se definen las variables a usar
    float A, B, C;
    float R;
    float X1, X2, Real, Imag;
    //Paso 1: Se piden los valores de A, B y C
    cout << "A = "; cin >> A;
    cout << "B = "; cin >> B;
    cout << "C = "; cin >> C;
    //Paso 2: Se calcula el radical
    R = B * B - 4 * A * C;
    //Paso 3: Preguntar por el valor del radical
    if (R < 0)
    {
         //Paso 4: Calcular el valor de la parte real
         //        y de la parte imaginaria.
         Real = -B / (2 * A);
         Imag = sqrt(-R) / (2 * A);
         //Imprimir las raices imaginarias
         printf("X1 = %f + %f i\nX2 = %f - %f i\n", Real, Imag, Real, Imag);
    }
    else
    {
         //Paso 7: Calcular las raices reales
         X1 = (-B + sqrt(R)) / (2 * A);
         X2 = (-B - sqrt(R)) / (2 * A);
         printf("X1 = %f\nX2 = %f", X1, X2);
    }
}
```

Decisión múltiple: (`switch`)

La mayoría de los lenguajes de programación manejan la toma de decisiones múltiples. En esencia, consiste en un procedimiento que hace posible elegir diferentes salidas, conforme al valor de una variable o expresión. La Figura 2.21 muestra un caso de esta instrucción.

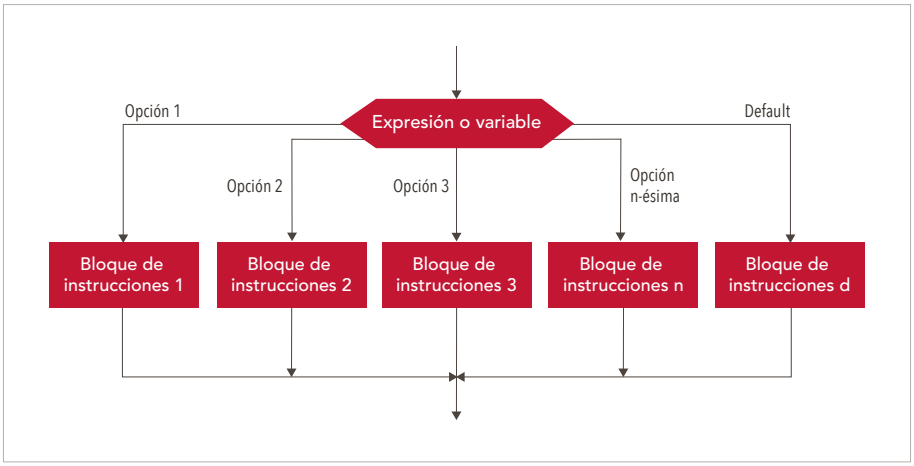


Figura 2.21. Bloque de decisión múltiple.

En C/C++ la cláusula por defecto (`default`) es opcional. Su función es disponer de cualquier valor de la expresión que discrepe con los casos definidos. Cada valor es representado por una opción de salida. En el Código 2.21 puede contemplarse la sintaxis correspondiente. Asimismo, el comando `break` se efectúa para interrumpir el cumplimiento de un bloque de decisión múltiple. En la Figura 2.22 se expone un ejemplo del uso de un bloque de opción múltiple. Se solicita un valor para la variable `o` y, posteriormente, se ejecuta el bloque correspondiente al valor ingresado. Si el valor de `o` está fuera del intervalo [1, 4], entonces se lleva a cabo la opción por defecto (`default`) y a `k` se le asigna el valor 0.

Código 2.21. Sintaxis del bloque de opción múltiple (`switch`).

```
switch (variable o expresión)
{
   case opción 1: //Bloque de instrucciones
        break;//Provoca la salida del switch
   case opción 2: //Bloque de instrucciones
        break;//Provoca la salida del switch
   case opción 3: //Bloque de instrucciones
        break;//Provoca la salida del switch
   case opción n: //Bloque de instrucciones
        break;//Provoca la salida del switch
   default:
        //Bloque de instrucciones
}
```

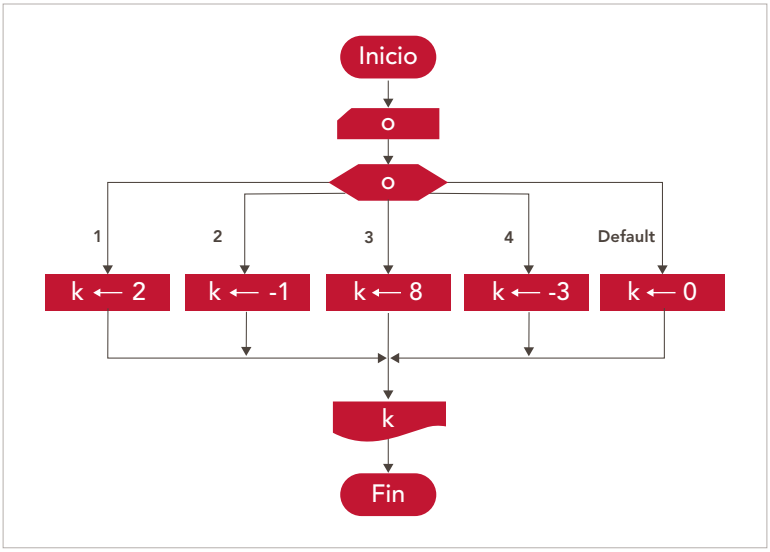


Figura 2.22. Ejemplo de uso del bloque de decisión múltiple.

El Código 2.22 exhibe el programa correspondiente al diagrama de flujo de la Figura 2.22.

Código 2.22. Ejemplo del uso del bloque `switch`.

```
#include <iostream>
using namespace std;
void main()
{
    short o;
    int k = 5;

    cout << "Dé un valor para o: ";
    cin >> o;//Se da un valor para la variable o
    //Implementación de la decisión múltiple
    switch (o)
    {
    case 1:
         cout << "Opción 1." << endl;
         k = 2;
         break;
    case 2:
         cout << "Opción 2." << endl;
         k = -1;
         break;
    case 3:
         cout << "Opción 3." << endl;
         k = 8;
         break;
```

```
    case 4:
         cout << "Opción 4." << endl;
         k = -3;
         break;
    default:
         cout << "Se ejecuta la opción por defecto." << endl;
         k = 0;
    }
    cout << "El valor de k es: " << k;
}
```

Al momento de ejecutar, se asigna a `o` el valor de 3, y se atribuye a `k` el valor de 8, como se ilustra a continuación:

```
Dé un valor para o: 3
Opción 3.
El valor de k es: 8
```

En caso de asignar a `o` un valor fuera del rango [1, 4] el resultado es el siguiente:

```
Dé un valor para o: 8
Se ejecuta la opción por defecto.
El valor de k es: 0
```

Al estudiante podrían surgirle dudas, como ejemplo, ¿qué ocurre si se omite el **`break`** en el **`switch`**? En el Código 2.23 se aprecia que los casos 1 y 2 carecen de **`break`**. Si la variable `o` vale 0, entonces se ejecuta el bloque del caso (**`case`**) 1, asignando a `k` el valor de 2. Al no existir **`break`**, se continúa con la ejecución del siguiente comando, que es asignar a `k` el valor de -1, correspondiente al caso 2. Como este tampoco tiene **`break`**, entonces se procede al caso 3, asignando a `k` el valor de 8.

Código 2.23. Uso del `switch` sin `break` en algunas opciones de salida.

```
#include <iostream>
using namespace std;
void main()
{
    short o;
    int k = 5;
    cout << "Dé un valor para o: ";
```

```
    cin >> o;//Se da un valor para la variable o
    //Implementación de la decisión múltiple
    switch (o)
    {
    case 1:
         cout << "Opción 1." << endl;
         k = 2;
    case 2:
         cout << "Opción 2." << endl;
         k = -1;
    case 3:
         cout << "Opción 3." << endl;
         k = 8;
         break;
    case 4:
         cout << "Opción 4." << endl;
         k = -3;
         break;
    default:
         cout << "Se ejecuta la opción por defecto." << endl;
         k = 0;
    }
    cout << "El valor de k es: " << k;
}
```

Resultado de la ejecución:

```
Dé un valor para o: 1
Opción 1.
Opción 2.
Opción 3.
El valor de k es: 8
```

Asimismo, la opción por defecto puede omitirse, como se muestra en el código:

```
#include <iostream>
using namespace std;
void main()
{
    short o;
    int k=5;

    cout << "Dé un valor para o: ";
    cin >> o;//Se da un valor para la variable o
    //Implementación de la decisión múltiple
    switch (o)
    {
    case 1:
         cout << "Opción 1." << endl;
         k = 2;
```

```
    case 2:
         cout << "Opción 2." << endl;
         k = -1;
    case 3:
         cout << "Opción 3." << endl;
         k = 8;
         break;
    case 4:
         cout << "Opción 4." << endl;
         k = -3;
         break;
    }
    cout << "El valor de k es: " << k;
}
```

Al ejecutarlo y dar a `o` un valor de 10, el resultado es el siguiente:

```
Dé un valor para o: 10
El valor de k es: 5
```

Antes de enviar a impresión, se debe asignar un valor a la variable `k`. De lo contrario, se generará una excepción, puesto que la variable no puede emplearse sin previa inicialización.

2.3.9. CICLOS, REPETICIONES O BUCLES

Se emplean cuando es necesario ejecutar una serie de instrucciones de manera reiterativa, cada vez que se cumpla una condición especifica. A continuación, se describe cada uno de los ciclos básicos.

Ciclo `for`

Tal comando se utiliza para repetir un bloque de código asignando un valor inicial a una o múltiples variables. Siempre que la condición se cumpla, las variables pueden incrementarse en cada iteración. En la Figura 2.23 se ilustra, mediante un diagrama, el flujo de funcionamiento de un ciclo. El mismo comienza con la asignación de valores iniciales a las variables definidas por el programador. En segundo lugar, se evalúa la condición: si la condición se cumple, se ejecuta el bloque de instrucciones correspondientes; en caso contrario, el ciclo

finaliza. Cabe destacar que el bloque de instrucciones únicamente se ejecutará mientras la condición se mantenga verdadera.

El ciclo **for** se constituye de tres partes (aunque cualquiera de ellas puede omitirse, inclusive todas): la asignación de valores iniciales a las variables, la condición que establece la permanencia en el ciclo y el incremento de las variables correspondientes. A continuación, se presentan algunas formas de trabajar un ciclo. No obstante, la sintaxis del **for** mostrado en la Figura 2.23 corresponde a una estructura general, y dependerá de cada lenguaje de programación su propio modo de ejecutarla.

```
for(valores iniciales; condición; incremento de variables).
        Inicio del ciclo.
        Bloque de instrucciones.
Fin del ciclo.
```

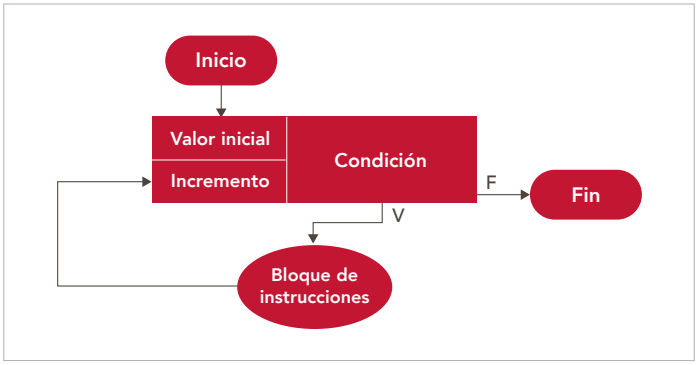


Figura 2.23. Esquema del ciclo for.

Ejercicio: Imprimir los números del 0 al 9

Comprensión del problema

Se requiere imprimir los números del 0 al 9. Para lograrlo, se propone una solución mediante la implementación de un ciclo. La salida esperada es la siguiente:

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

Algoritmo

Ahora, se muestra el algoritmo general para resolver el problema.

Algoritmo 2.5. Imprimir los números del 0 al 9.

```
1. El ciclo comienza asignando a la variable i el valor de 0.
2. Se verifica si la condición es verdadera (V) o falsa (F).
3. Si la condición i < 10 es verdadera, se imprime el valor de i.
4. Se incrementa en 1 el valor de i.
5. Se regresa al Paso 2 para comprobar si la condición i < 10 aún se cumple.
6. Mientras la condición sea verdadera, se repiten los pasos 3 y 4. Cuando
sea falsa, termina la ejecución del programa.
```

Diagrama de flujo

La Figura 2.24 expone el diagrama de flujo que corresponde a la impresión de los valores del 0 al 9.

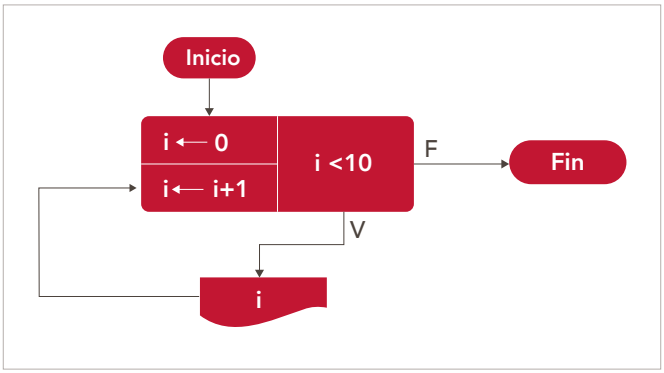


Figura 2.24. Uso de un ciclo for para imprimir los números del 0 al 9.

Código en C

El programa correspondiente en lenguaje C (Figura 2.24) se presenta en el Código 2.24. Sin embargo, existen diversas formas de escribir este programa para que realice la misma acción, como se muestra en la Tabla 2.16.

Código 2.24. Imprimir los números enteros del 0 al 9.

```
#include <iostream>
void main()
{
    int i;
    for (i = 0; i < 10; i++)  printf("%d\t", i);
}
```

Tabla 2.16. Opciones de código para el ciclo.

OPCIÓN 1	`int i;`
	`for (i = 0; i < 10; i++)`
	`  printf("%d\t", i);`

OPCIÓN 2	`int i;`
	`for (i = 0; i < 10; i++)`
	`{`
	`  printf("%d\t", i);`
	`}`

OPCIÓN 3	`for (int i = 0; i < 10; i++)` `  printf("%d\t", i);`
OPCIÓN 4	`for (int i = 0; i <= 9; i++)` `  printf("%d\t", i);`

La Figura 2.25 ilustra el proceso de ejecución de un ciclo **`for`**. En un principio, el ciclo procede del bloque de inicialización de variables; enseguida, se evalúa la condición para determinar si se entra o se sale del ciclo. Si la condición se satisface, se procede a ejecutar el bloque de instrucciones del ciclo. De lo contrario, el ciclo finaliza. Una vez completadas las instrucciones, se procede al bloque de incrementos y se vuelve a evaluar la condición.

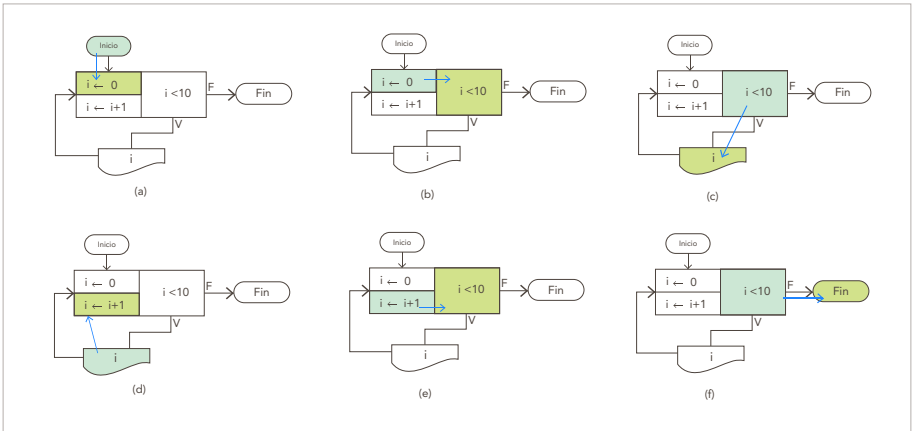


Figura 2.25. (a) Paso 1. (b) Paso 2. (c) Paso 3, mientras la condición se cumpla. (d) Paso 4. (e) Retorno al Paso 2. (f) Paso 6, cuando la condición se incumpla.

Ejercicio: Imprimir la tabla de multiplicar del 2

Comprensión del problema

Se pretende que el programa genere como salida la tabla de multiplicar del número 2, justo como se muestra en la Tabla 2.17.

Tabla 2.17. Tabla de multiplicar del 2.

```
1 × 2 = 2
2 × 2 = 4
3 × 2 = 6
4 × 2 = 8
5 × 2 = 10
6 × 2 = 12
7 × 2 = 14
8 × 2 = 16
9 × 2 = 18
10 × 2 = 20
```

Algoritmo

A continuación, se muestra el Algoritmo 2.6, que corresponde al método para desplegar en pantalla la tabla del 2.

Algoritmo 2.6. Tabla de multiplicar del 2 utilizando un ciclo for.

```
1. Se definen las variables.
2. for(se inicia i en 1; hasta 10; incrementa en 1).
3. Inicio del ciclo.
4.      Se multiplica 2 por i.
5.      Se imprime el resultado.
6. Fin del ciclo.
```

Diagrama de flujo

La Figura 2.26 muestra el diagrama de flujo correspondiente al Algoritmo 2.6.

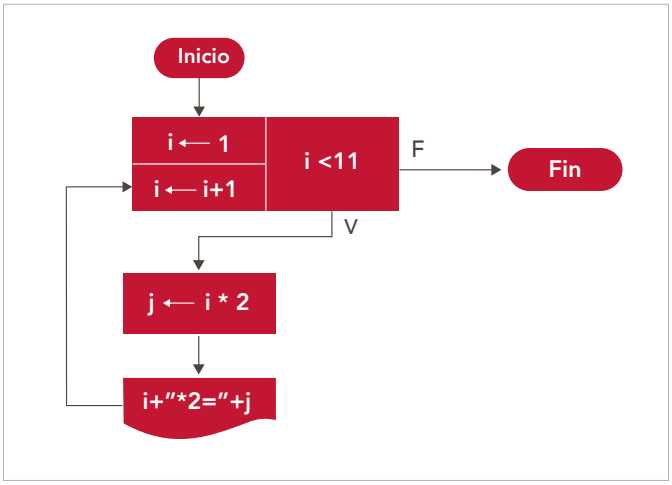


Figura 2.26. Diagrama de flujo para la tabla de multiplicar del 2.

Código en C

Al final, el programa en lenguaje C (Figura 2.26) se muestra en el Código 2.25.

Código 2.25. Imprimir la tabla de multiplicar del 2.

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

void main()
{
    int i, j;

    for (i = 1; i <= 10; i = i + 1)
    {
         j = i * 2;
         printf(" % d * 2 = % d\n", i, j);
    }

    _getch();
}
```

Ejercicio: Imprimir los primeros *N* valores de la serie de Fibonacci

Comprensión del problema

La serie de Fibonacci se caracteriza por ser una secuencia infinita de números naturales que comienza con 0 y 1, y donde cada número subsecuente es la suma de los dos anteriores. Los primeros ocho términos de la serie son:

$$0, 1, 1, 2, 3, 5, 8, 13$$

Es decir, la secuencia puede representarse mediante la siguiente expresión de modo que se asimile a la mostrada en la Figura 2.27:

$$D_i = D_{i-1} + D_{i-2} \;\; para\; i > 2$$

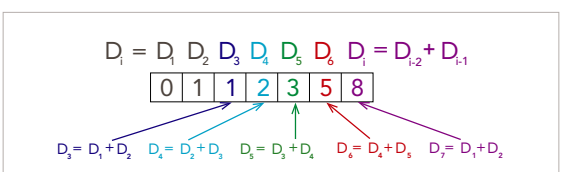


Figura 2.27. Serie de Fibonacci.

Algoritmo

El algoritmo para calcular e imprimir los términos de la serie de Fibonacci se presenta a continuación. El valor de N representa el número de términos de la secuencia a imprimir, y debe ser mayor que 0. Si $N = 1$, se imprime el 0; si $N = 2$, se imprimen los valores 0 y 1.

Algoritmo 2.7. Imprimir N términos de la serie de Fibonacci.

```
1. Se definen las variables A, B, y C.
2. Se inicializan las variables A y B. A = 0, B = 1.
3. Se pide el valor de N.
4. ¿N > 0? Sí: Ir al paso 5. No: Ir al paso 13.
5. Imprimir el valor 0.
6. ¿N > 1? Sí: Ir al paso 7. No: Ir al paso 8.
```

```
7. Imprimir el valor 1.
8. for(se inicia i en 3; hasta N; incrementa en 1).
9. Inicio del ciclo.
10.    Se asigna a C el valor de A + B.
11.    Imprimir C.
12.    Asignar a A el valor de B.
13.    Asignar a B el valor de C.
14. Fin del ciclo.
15. Fin del programa.
```

Diagrama de flujo

El diagrama correspondiente se muestra en la Figura 2.28.

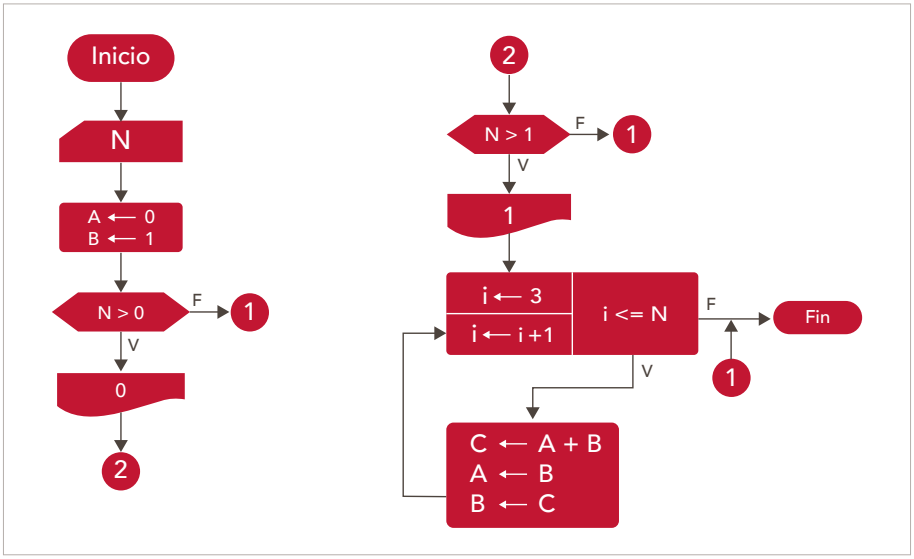


Figura 2.28. Imprimir *N* elementos de la serie de Fibonacci.

Código en C

El programa para obtener los *N* elementos de la serie de Fibonacci se presenta en el Código 2.26. En principio, `scanf` es una función de entrada de datos que se emplea normalmente para este tipo de aplicaciones; otros compiladores la soportan sin restricciones, sin embargo, Visual Studio 2019 la considera insegura. Por tanto, en este caso, se usa `scanf_s`.

Código 2.26. Imprimir *N* elementos de la serie de Fibonacci.

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

void main()
{
    int N, A, B, C;
    int i;
    A = 0;
    B = 1;
    printf("Número de elementos deseados: ");
    scanf_s("%d", &N);
    if (N > 0)
    {
         printf("0\t");
         if (N > 1)
         {
              printf("1\t");
              for (i = 3; i <= N; i++)
              {
                   C = A + B;
                   printf("%d\t", C);
                   A = B;
                   B = C;
              }
         }
    }
    _getch();
}
```

Ejercicio: Ciclos anidados para imprimir las tablas del 1 al 5

Comprensión del problema

Se busca imprimir las tablas de multiplicar del 1 al 5. La salida esperada del programa corresponde a la que se expone a continuación.

1*1 = 1	1*2 = 2	1*3 = 3	1*4 = 4	1*5 = 5
2*1 = 2	2*2 = 4	2*3 = 6	2*4 = 8	2*5 = 10
3*1 = 3	3*2 = 6	3*3 = 9	3*4 = 12	3*5 = 15
4*1 = 4	4*2 = 8	4*3 = 12	4*4 = 16	4*5 = 20
5*1 = 5	5*2 = 10	5*3 = 15	5*4 = 20	5*5 = 25
6*1 = 6	6*2 = 12	6*3 = 18	6*4 = 24	6*5 = 30
7*1 = 7	7*2 = 14	7*3 = 21	7*4 = 28	7*5 = 35
8*1 = 8	8*2 = 16	8*3 = 24	8*4 = 32	8*5 = 40
9*1 = 9	9*2 = 18	9*3 = 27	9*4 = 36	9*5 = 45
10*1 = 10	10*2 = 20	10*3 = 30	10*4 = 40	10*5 = 50

Algoritmo

Para resolver el ejercicio es necesario manipular dos variables: la primera controla el número de la tabla y la segunda el factor de multiplicación. Esta lógica se aplica en el Algoritmo 2.8.

Algoritmo 2.8. Imprimir tablas de multiplicar del 1 al 5.

```
1. Se definen las variables A, B y C.
2. for(se inicia A en 1; hasta 10; incrementa en 1).
3. Inicio del ciclo.
4. for(se inicia B en 1; hasta 5; incrementa en 1).
5.      Inicio del ciclo.
6.              Se asigna a C el valor de A * B.
7.              Se imprime A + “*” + B + “=” + C.
8.      Fin del ciclo.
9. Fin del ciclo.
10. Fin del programa.
```

Diagrama de flujo

El diagrama se representa en la Figura 2.29.

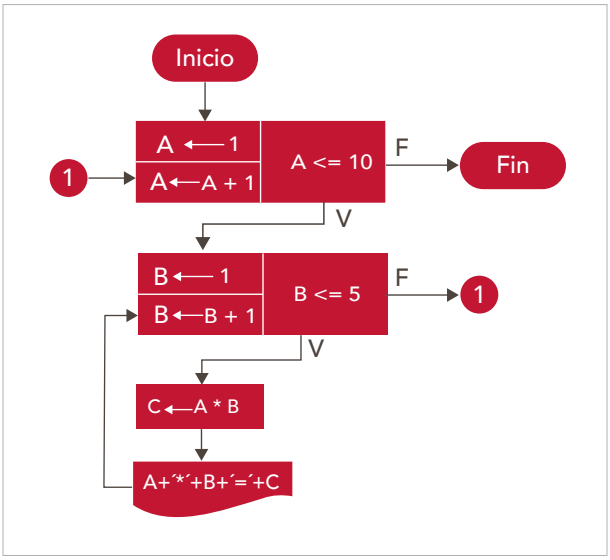


Figura 2.29. Imprimir la tabla de multiplicar del 1 al 5.

Código en C

El programa correspondiente se muestra en el Código 2.27.

Código 2.27. Imprimir las tablas de multiplicar del 1 al 5.

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

int A, B, C;

void main()
{
   for (A = 1; A <= 10; A++)
   {
        for (B = 1; B <= 5; B++)
        {
             C = A * B;
             printf("%d*%d=%d\t\t", A, B, C);
        }
        printf("\n");
   }
   _getch();
}
```

Ciclo `do-while`

Se trata de otra estructura utilizada para ejecutar de forma repetitiva un bloque de instrucciones. Puede interpretarse como: "ejecuta (`do`) ... bloque de instrucciones ... mientras (`while`) ...". La característica principal de este ciclo es que garantiza la ejecución del bloque de instrucciones al menos en una ocasión. El Algoritmo 2.9 presenta una manera de interpretar este tipo de ciclo; la Figura 2.30, un diagrama de flujo del ciclo `while`.

Algoritmo 2.9. Esquema general de un ciclo `do-while`.

```
1. do.
2. Inicio del bloque de instrucciones.
3.      bloque de instrucciones a realizar mientras se cumpla la condición.
4. Fin del bloque de instrucciones.
5. while (condición).
```

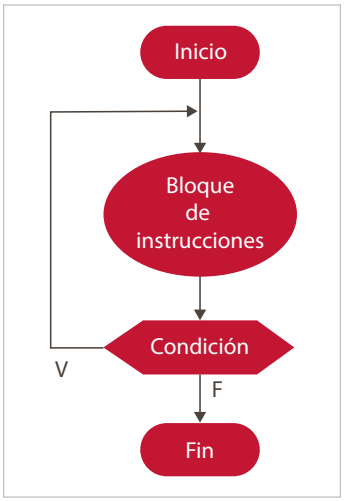


Figura 2.30. Diagrama de flujo de un ciclo `do-while`.

Ejercicio: Imprimir la tabla de multiplicar del 5 con ciclo `do-while`

Comprensión del problema

Se solicita generar la tabla de multiplicar del 5, por tanto, la salida esperada es la siguiente:

```
1*5 = 5
2*5 = 10
3*5 = 15
4*5 = 20
5*5 = 25
6*5 = 30
7*5 = 35
8*5 = 40
9*5 = 45
10*5 = 50
```

Algoritmo

El Algoritmo 2.10 presenta la estructura general para desplegar la tabla de multiplicar del número 5 mediante un ciclo **`do-while`**.

Algoritmo 2.10. Estructura de un ciclo `do-while` para la tabla de multiplicar del 5.

```
1. Inicia i en 1.
2. do.
3.       Inicio del ciclo.
4.                Multiplicar i por 5 y guardar el resultado en j.
5.                Imprimir i, " * 5 = ", j.
6.                Incrementar i.
7.       Fin del ciclo.
8. while(i sea menor o igual a 10)
9. Fin del programa.
```

Diagrama de flujo

La Figura 2.31 muestra el diagrama de flujo correspondiente, usando un ciclo **`do-while`** en lugar de un ciclo **`for`**.

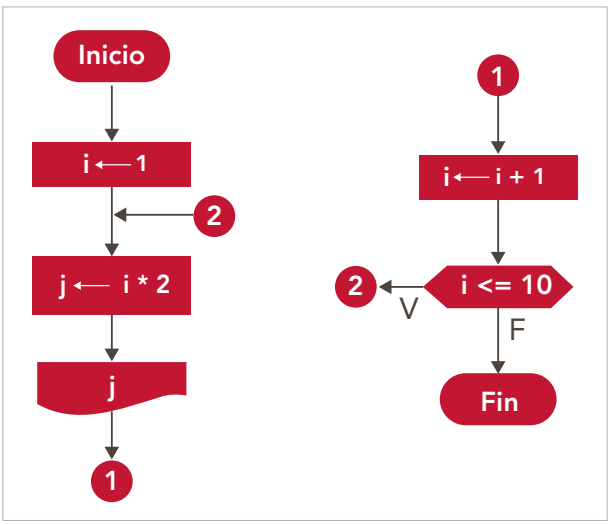


Figura 2.31. Diagrama de flujo para imprimir la tabla de multiplicar del 2.

Código en C

El programa en lenguaje C se presenta en el Código 2.28.

Código 2.28. Programa en lenguaje C para imprimir la tabla de multiplicar del 5.

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

int i = 1, j;

void main()
{
    do
    {
         j = i * 5;
         printf("%d * 5 = %d\n", i, j);
         i++;
    } while (i <= 10);
}
```

Ejercicio: Calcular el factorial de un número *N*

Comprensión del problema

El factorial de un número se expresa mediante:

$$N! = 1 \times 2 \times 3 \times \cdots \times N$$

En otras palabras, el cálculo del factorial de un número N consiste en multiplicar de manera consecutiva todos los enteros positivos desde 1 hasta N. Esta operación es fundamental en combinatoria y probabilidad para calcular el número de maneras de organizar los elementos de un conjunto finito. Entonces, se presentan los siguientes ejemplos:

$$0! = 1$$

$$1! = 1$$

$$2! = 1 \times 2 = 2$$

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

$$6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$$

$$10! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10 = 3628800$$

El cálculo del factorial solo es válido para números enteros y positivos, y puede sintetizarse a través de la siguiente fórmula:

$$F = \prod_{i=1}^{N} i$$

Donde:

- Π operación de multiplicación sucesiva.
- i índice que toma valores desde 1 hasta N.
- F resultado del cálculo factorial.

Dado lo anterior, se requiere una variable N que indique el número del cual se calculará el factorial, una F que almacene el acumulado de las multiplicaciones y una variable i que incremente desde 1 hasta N.

Algoritmo

Para calcular el factorial del valor N se propone el Algoritmo 2.11.

Algoritmo 2.11. Calcular el factorial de N.

```
1. Pedir el valor de N.                                              N ← ?
2. Iniciar una variable F en 1.                                      F ←1
3. Iniciar una variable i en 1.                                      i ←1
4. Se realiza la operación F ∗ i y se almacena el resultado en F.    F ←F∗i
5. Se incrementa i en 1.                                             i ←i+1
6. ¿Es i ≤ N? Sí: Ir al paso 7. No: Ir al paso 4.
7. Imprimir el valor de F.
8. Fin.
```

Diagrama de flujo

La Figura 2.32 exhibe el diagrama de flujo que utiliza un ciclo **`do-while`** en sustitución de un ciclo **`for`** para realizar la misma función.

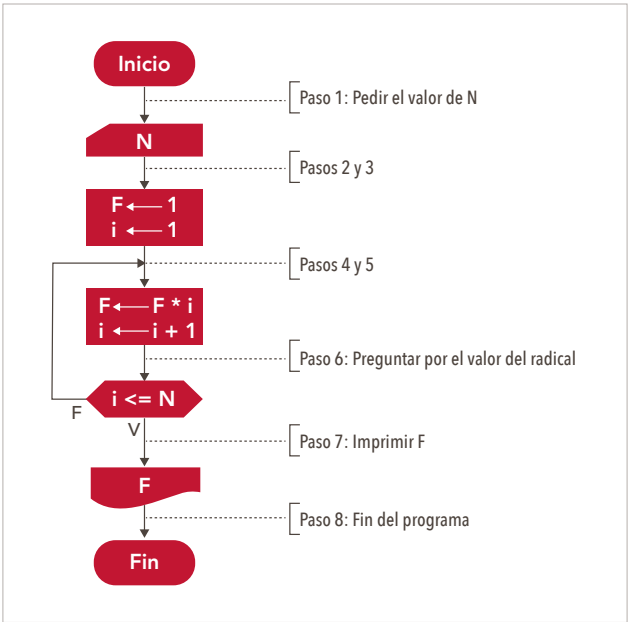


Figura 2.32. Diagrama de flujo para calcular el factorial de un número *N*.

Código en C

El programa en lenguaje C que corresponde al diagrama de la Figura 2.32 es mostrado en el Código 2.29. Las líneas pueden simplificarse, como se aprecia en el Código 2.30, no obstante, con el inconveniente de que se obvian algunos pasos en su ejecución.

Código 2.29. Calcular el factorial de un número *N*.

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

void main()
{
    int i, F;
    int N;

    scanf_s("%d", &N);//Paso 1
    F = 1;//Paso 2
    i = 1;//Paso 3
    do
```

```
    {
        F = F * i;//Paso 4
        i = i + 1;//Paso 5
    } while (i ≤ N); //Paso 6
    printf("Factorial de %d es %d", N, F); //Paso 7
}
```

Código 2.30. Programa simplificado para calcular el factorial de *N*.

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

void main()
{
    int i = 1, F = 1, N;

    scanf_s("%d", &N);
    i = F = 1;

    do
    {
        F *= i;
        i++;
    } while (i ≤ N); //Paso 6
    printf("Factorial de %d es %d", N, F);
}
```

Ciclo `while`

Este ciclo ejecuta un bloque de instrucciones mientras (**`while`**) una condición especificada se cumpla. Tanto **`while`** como **`do-while`** resultan útiles cuando el número de repeticiones de una instrucción es variable. Con respecto de **`while`**, es posible que el bloque se omita si la condición resulta incumplida desde el inicio. El Algoritmo 2.12 muestra la estructura general de un ciclo **`while`**.

Algoritmo 2.12. Estructura general de un ciclo `while`.

```
1. while (condición).
2. begin.
3.      bloque de comandos o instrucciones.
4. end.
```

La Figura 2.33 esquematiza paso por paso el diagrama de flujo correspondiente al Algoritmo 2.12 de un ciclo **`while`**.

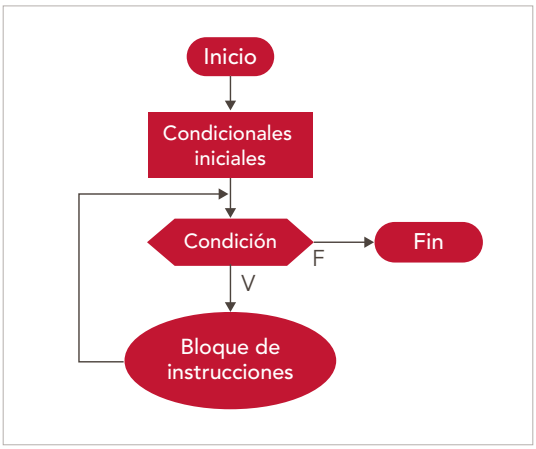


Figura 2.33. Diagrama de flujo para un ciclo `while`.

Ejercicio: Tabla de multiplicar del número 7

Comprensión del problema

A fin de continuar con el ejemplo de las tablas de multiplicar, ahora se utilizará el ciclo **`while`** para generar la tabla del número 7, de modo que el resultado en pantalla sea el siguiente:

```
1*7 = 7
2*7 = 14
3*7 = 21
4*7 = 28
5*7 = 35
6*7 = 42
7*7 = 49
8*7 = 56
9*7 = 63
10*7 = 70
```

Algoritmo

El Algoritmo 2.13 se ostenta como una propuesta metodológica para implementar los pasos necesarios en la construcción de la tabla del 7.

Algoritmo 2.13. Algoritmo para la tabla del 7.

```
1. Inicia i en 1.
2. while (i sea menor o igual a 10).
3.      Inicio del ciclo.
4.              Multiplicar i por 7 y guardar el resultado en j.
5.              Imprimir i, “ * 7 = ”, j.
6.              Incrementar i.
7.      Fin del ciclo.
8. Fin.
```

Diagrama de flujo

La Figura 2.34 ilustra el diagrama de flujo correspondiente al ciclo **`while`**.

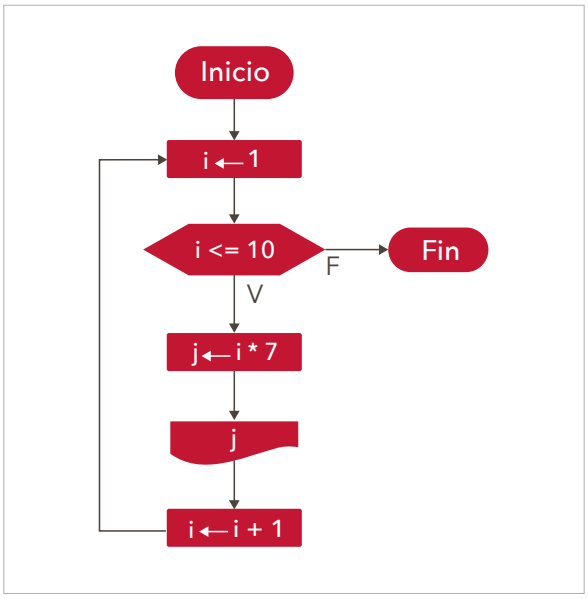


Figura 2.34. Diagrama de flujo para imprimir la tabla de multiplicar del 7 usando el ciclo `while`.

Código en C

El programa para dar solución al ejercicio se despliega línea por línea en el Código 2.31.

Código 2.31. Programa en C para imprimir la tabla de multiplicar del 7 usando el ciclo `while`.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int i = 1, j;

void main()
{
   while(i<=10)
   {
        j = i * 7;
        printf("%d * 7 = %d\n", i, j);
        i++;
   }
}
```

Diferencia entre los ciclos

La práctica es pieza clave para que el programador consolide lo aprendido y distinga en qué situaciones conviene utilizar cada tipo de ciclo, puesto que la diferencia entre los tres revisados es sutil, pero decisiva. Por un lado, **`do-while`** garantiza que el bloque de instrucciones se ejecutará al menos una vez. Por otro, en **`while`** es posible que se omita su corrida si la condición inicial resulta insatisfecha. En cuanto a **`for`**, este puede proceder de una forma similar a **`while`**, aunque ofrece una estructura más compacta para controlar iteraciones.

Comando `break`

Tal clave se aprovecha para interrumpir la ejecución de un bucle de instrucciones y se incorpora adentro de un bloque de opción múltiple. El Código 2.32 ejemplifica una instancia de la aplicación del **`break`** en un ciclo **`for`**.

Código 2.32. Uso de break en un ciclo.

```
#include <iostream>
#include <conio.h>
using namespace std;

void main()
{
   for (int i = -5; i ≤ 5; i++)
   {
        cout << i << "\t";
        if (!i)
             break;
   }
   _getch();
}
```

Al ejecutar el programa del Código 2.32, se observa que mientras la condición `!i` sea incumplida, el ciclo continúa ejecutándose e imprime el valor de `i`. En caso de cumplirse la condición, se activa el **`break`** y el ciclo se interrumpe:

```
-5      -4      -3      -2      -1      0
```

La instrucción **`break`** también puede emplearse en los ciclos **`while`** y **`do-while`**. En principio, su uso es similar al que tiene un ciclo **`for`**.

2.3.10. CONSTANTES Y MACROS

En todo lenguaje de programación existe la definición de constantes. Tales se emplean para asignar valores que permanecen inmodificables en el programa. La sintaxis se muestra enseguida; asimismo, se puede notar la omisión de "**;**" al final de la línea.

```
#define <nombre> valor
```

Para ejemplificar el uso de constantes en C, se desarrollará un programa que calcula el área de un círculo, con base en el diagrama de flujo de la Figura 2.35. En el Código 2.33 se muestra cómo utilizar constantes, definiendo el valor de la constante **`pi`** como 3.1416, el cual se toma para realizar el cálculo del área.

Código 2.33. Uso de una constante para definir el valor de π.

```
#include <iostream>
#define pi 3.1416
using namespace std;

int main()
{
   float r, a;

   r = (float)2.0;
   a = (float)(pi * r * r);
   cout << a;
}
```

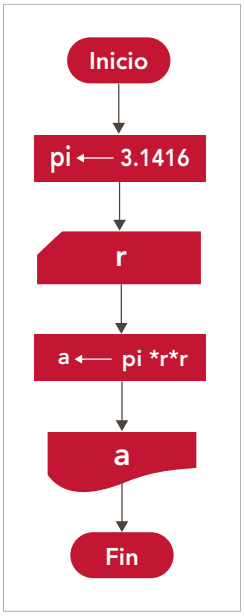


Figura 2.35. Diagrama para el cálculo del área de un círculo.

Por su parte, los macros son instrucciones definidas de manera abreviada y se declaran con la directiva **`#define`**. Un manejo adecuado de los macros permite simplificar el código de un programa; se aplican por medio de la siguiente sintaxis:

```
#define <nombre de la función> <línea de comandos>
```

En el Código 2.34 se aprecia el uso de un macro para calcular el área de un círculo, donde `x` representa la longitud del radio. El diagrama de flujo corresponde al de la Figura 2.35; lo que cambia es la implementación: al utilizar macros, se aprovechan las ventajas del lenguaje y se simplifica el código.

Código 2.34. Cálculo del área de un círculo por medio de macros.

```
#include <iostream>
#define pi 3.1416
#define f(x) x*x*pi
using namespace std;
void main()
{
   float r, a;

   r = (float)2.0;
   a = f(r);
   cout << a;
}
```

Otro ejemplo corresponde al cálculo de las raíces de una ecuación cuadrática, ilustrado en el diagrama de flujo de la Figura 2.36.

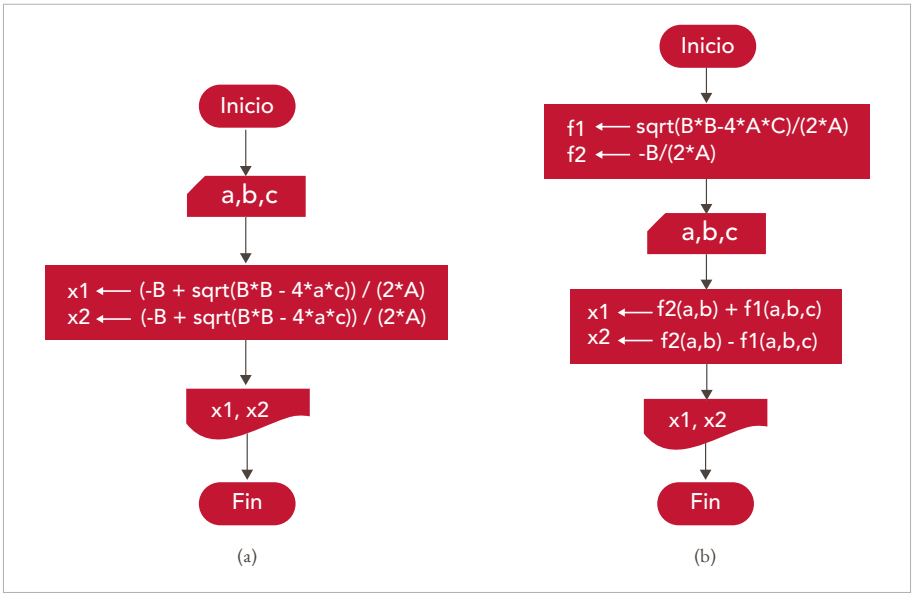


Figura 2.36 Solución de la ecuación cuadrática: (a) Sin macros. (b) Con macros.

El programa en lenguaje C correspondiente a la Figura 2.36(a) se presenta en el Código 2.35, mientras que el de la Figura 2.36(b) se muestra en el Código 2.36.

Código 2.35. Solución de la ecuación cuadrática sin macros.

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

using namespace std;

void main()
{
   double a, b, c;
   double x1, x2;

   a = 12.0; b = -36.0; c = -120.0;
   x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
   x2 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);
   cout << x1 << endl << x2 << endl;
}
```

Código 2.36. Solución de la ecuación cuadrática con macros, opción 1.

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

#define f1(A,B,C) sqrt(B*B-4*A*C)/(2*A)

#define f2(A,B)   -B/(2*A)

using namespace std;

void main()
{

   double a, b, c;
   double x1, x2;

   a = 12.0; b = -36.0; c = -120.0;
   x1 = f2(a, b) + f1(a, b, c);
   x2 = f2(a, b) - f1(a, b, c);

   cout << x1 << endl << x2 << endl;
}
```

En la práctica la cantidad de macros a utilizar es ilimitada. Una variante del Código 2.36 se muestra en el Código 2.37, donde se emplean diversos macros para resolver la ecuación cuadrática. En resumen, se puede seguir el mismo diagrama de flujo, salvo la variante de que se incorporan macros para simplificar su implementación.

Código 2.37. Solución de la ecuación cuadrática con macros, opción 2.

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

#define f1(A,B,C) sqrt(B*B-4*A*C)/(2*A)
#define f2(A,B)   -B/(2*A)
#define g1(A,B,C) f2(A,B)+f1(A,B,C)
#define g2(A,B,C) f2(A,B)-f1(A,B,C)

using namespace std;

void main()
{
    double a, b, c;
    double x1, x2;
    a = 12.0; b = -36.0; c = -120.0;
    x1 = g1(a, b, c);
    x2 = g2(a, b, c);
    cout << x1 << endl << x2 << endl;
}
```

De igual forma, es posible introducir la operación del cálculo de la raíz en un macro, como se observa en el Código 2.38.

Código 2.38. Código simplificado para la ecuación cuadrática con macros.

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

#define f1(A,B,C) (-B+sqrt(B*B-4*A*C))/(2*A)
#define f2(A,B,C) (-B-sqrt(B*B-4*A*C))/(2*A)
using namespace std;

void main()
{
    double a, b, c;
    double x1, x2;
    a = 12.0; b = -36.0; c = -120.0;
    x1 = f1(a, b, c);
    x2 = f2(a, b, c);
    cout << x1 << endl << x2 << endl;
}
```

2.3.11. ARREGLOS Y CADENAS DE CARACTERES

Hasta el momento, se han utilizado variables para almacenar un único dato. No obstante, en la práctica suele requerirse manejar múltiples valores dentro de una sola variable. Para ello existen los arreglos, que pueden ser unidimensionales (vectores) o bidimensionales (matrices). La Figura 2.37(a) ilustra una matriz, mientras que la Figura 2.37(b) muestra un vector.

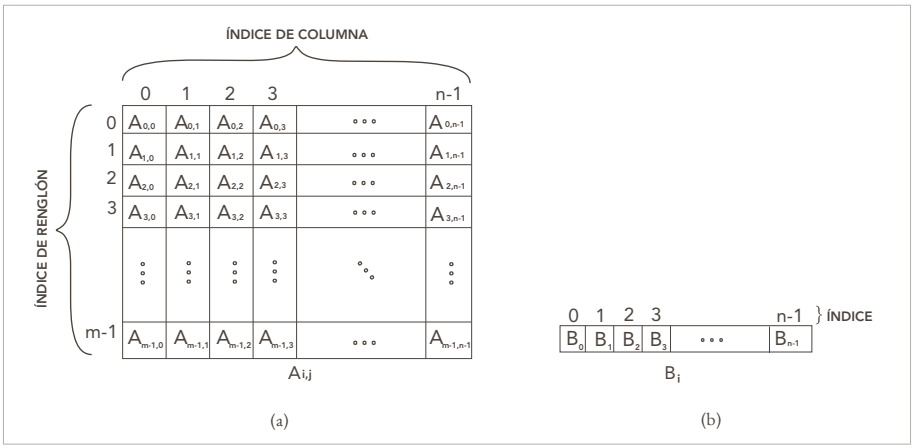


Figura 2.37. Estructura de un arreglo. (a) Bidimensional. (b) Unidimensional.

En la Figura 2.37(a), el arreglo bidimensional se representa mediante la variable *A*, y maneja dos índices: *i* y *j*. El índice *i* corresponde al número de renglón, con valores de 0 hasta *m* - 1, mientras el índice *j* es para el número de columna, con valores de 0 hasta *n* - 1. En la Figura 2.37(b) el vector se identifica con la variable *i*, cuyo rango de valores va de 0 hasta *n* - 1, y *n* representa el número total de elementos almacenados en el arreglo. Cabe destacar, en algunos lenguajes de programación, el índice no inicia en 0, sino en 1. Por ello, es importante que el programador verifique el valor mínimo permitido por el lenguaje que esté utilizando.

Arreglos unidimensionales

El Algoritmo 2.14 puede implementarse de diversas maneras. Se muestra en los diagramas de flujo cómo se desenvuelven los ciclos `for`, `do-while` y `while`. El Código 2.39 corresponde a la Figura 2.38(a); así, el Código 2.40 es la programación del diagrama de la Figura 2.38(b), y finalmente, el Código 2.41 ilustra la perteneciente a la Figura 2.38(c).

Algoritmo 2.14. Asignación de datos a un vector.

```
1. Se define usar la variable A como el arreglo.
2. Se pide el número de datos y se almacena en n.
3. Inicia el índice i en 0.
4. Se pide el valor de Ai.
5. Se incrementa el valor de i en 1.
6. ¿Es i < n?
Sí: Ir al paso 4.
No: Ir al paso 7.
7. Fin.
```

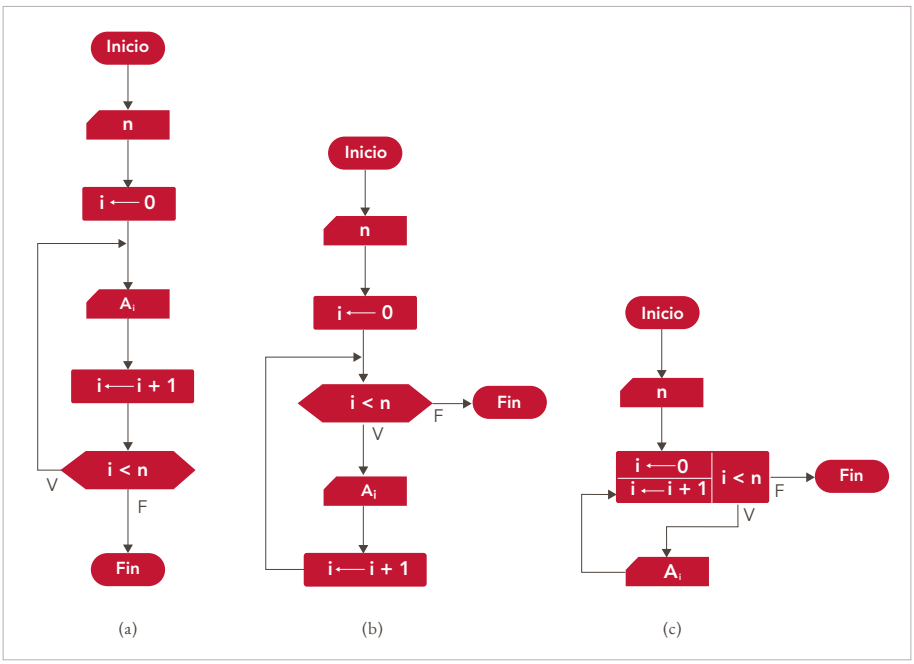


Figura 2.38. Asignación de datos a un vector. (a) `do-while`. (b) `while`. (c) `for`.

Código 2.39. Programa en C para asignar datos a un arreglo a través de `do-while`.

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

void main()
{
```

```
int n;  //Tamaño del arreglo
    int i;//indice del arreglo

    //Paso 1: Se define la variable A como arreglo
    int A[10];
    //Paso 2: Se pide el número de datos y se almacena en n
    printf("Tamaño del arreglo: "); scanf_s("%d", &n);
    //Paso 3: Inicia el índice i en 0
    i = 0;
    do
    {
         //Paso 4: Se pide el valor de Ai
         printf("A[%d]=", i); scanf_s("%d", &A[i]);
         //Paso 5: Se incrementa el valor de i en 1
         i++;
         //Paso 6: ¿Es i < n? Sí: ir al paso 4. No: ir al paso 7.
    } while (i < n);
    //Paso 7: Fin
}
```

Código 2.40. Asignación de datos a un arreglo mediante el ciclo `while`.

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

void main()
{
    int n;//Tamaño del arreglo
    int i;//Índice del arreglo

    //Paso 1: Se define la variable A como arreglo
    int A[10];
    //Paso 2: Se pide el número de datos y se almacena en n
    printf("Tamaño del arreglo: "); scanf_s("%d", &n);
    //Paso 3: Inicia el índice i en 0
    i = 0;
    //Paso 6: ¿Es i < n? Sí: ir al paso 4. No: ir al paso 7.
    while (i < n)
    {
         //Paso 4: Se pide el valor de Ai
         printf("A[%d]=", i); scanf_s("%d", &A[i]);
         //Paso 5: Se incrementa el valor de i en 1
         i++;
    }
    //Paso 7: Fin
}
```

Código 2.41. Asignación de datos a un arreglo vía el ciclo *for*.

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

void main()
{
   int n;//Tamaño del arreglo
   int i;//Índice del arreglo

   //Paso 1: Se define la variable A como arreglo
   int A[10];
   //Paso 2: Se pide el número de datos y se almacena en n
   printf("Tamaño del arreglo: "); scanf_s("%d", &n);
   //Pasos 3, 6 y 5 van en el ciclo for
   for (i = 0; i < n; i++)
   {
        //Paso 4: Se pide el valor de Ai
        printf("A[%d]=", i); scanf_s("%d", &A[i]);
   }
   //Paso 7: Fin
}
```

El Algoritmo 2.14 solicita exclusivamente los valores de un arreglo; sin embargo, con el fin de verificar que los datos se almacenaron correctamente, se incluyó en el Algoritmo 2.15 una etapa adicional: la impresión de valores. La Figura 2.39 muestra el diagrama de flujo correspondiente al Algoritmo 2.15, el cual puede implementarse con comandos **`while`**, **`do-while`** o **`for`**. La programación del flujo en la Figura 2.39(a) se ejemplifica en el Código 2.42, donde se emplea un bucle **`do-while`**. Por su parte, el Código 2.43 utiliza un ciclo **`for`**. Ambos enfoques son válidos; la elección del método dependerá particularmente de la experiencia y estilo del desarrollador.

Algoritmo 2.15. Asignación de datos a un vector e impresión de los valores.

1. Se define usar la variable A como el arreglo.
2. Se pide el número de datos y se almacena en n.
3. Inicia el índice i en 0.
4. Se pide el valor de A_i.
5. Se incrementa el valor de i en 1.
6. ¿Es i < n? Sí: Ir al paso 4. No: Ir al paso 7.
7. Inicia el índice i en 0.
8. Se imprime el valor de A_i.
9. Se incrementa el valor de i en 1.
10. ¿Es i < n? Sí: Ir al paso 8. No: Ir al paso 11.
11. Fin.

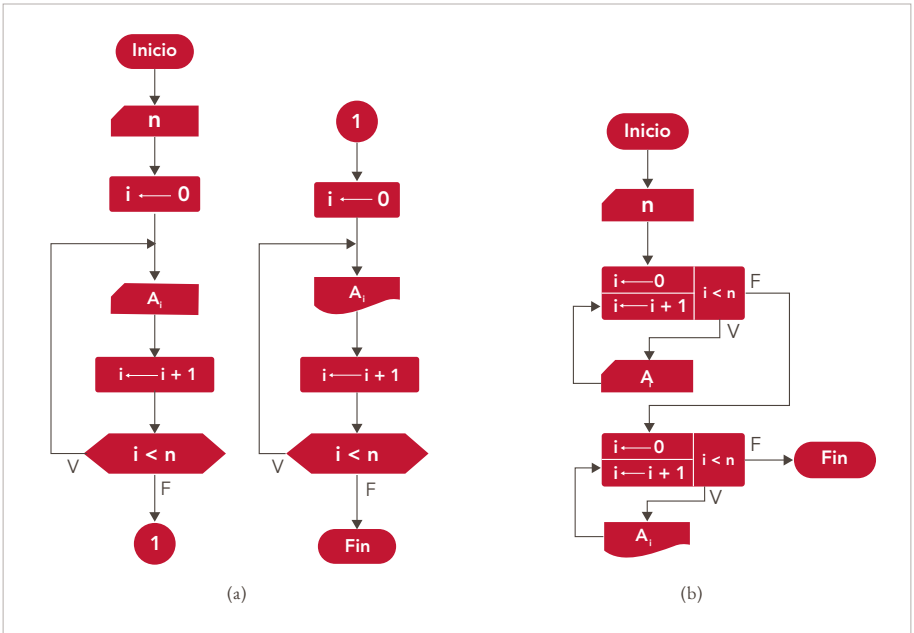


Figura 2.39. Diagrama de flujo para asignar e imprimir valores de un vector. (a) `do-while`. (b) `for`.

Existe otra manera de asignar datos a un arreglo, aunque la sintaxis depende del lenguaje de programación ejecutado. Tal técnica se denomina como *preasignación de valores* y consiste en inicializar los datos al momento de declarar el arreglo. Conviene revisar el Código 2.44, escrito en C y compilado en Visual Studio 2019. A este respecto, recurre al uso de la librería **`iostream`** para acceder a la función de impresión **`cout`**. Cabe puntualizar que, para emplear **`cout`**, es primordial incluir la línea **`using namespace std;`**, ya que es requisito específico de Visual Studio. En otras plataformas este paso es innecesario.

Código 2.42. Imprimir los valores de un vector usando un ciclo `do-while`.

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

void main()
{
   int n;//Tamaño del arreglo
   int i;//Índice del arreglo
```

```
//Paso 1: Se define la variable A como arreglo
   int A[10];
   //Paso 2: Se pide el número de datos y se almacena en n
   printf("Tamaño del arreglo: "); scanf_s("%d", &n);
   //Paso 3: Inicia el índice i en 0
   i = 0;
   do
   {
        //Paso 4: Se pide el valor de Ai
        printf("A[%d]=", i); scanf_s("%d", &A[i]);
        //Paso 5: Se incrementa el valor de i en 1
        i++;
        //Paso 6: ¿Es i < n? Sí: ir al paso 4. No: ir al paso 7.
   } while (i < n);
   //Paso 7: Inicia el índice en 0
   i = 0;
   do
   {
        //Paso 8: Se imprime el valor de Ai
        printf("A[%d]=%d\n", i, A[i]);
        //Paso 9: Se incrementa el valor de i en 1
        i++;
        //Paso 10: ¿Es i < n? Sí: ir al paso 8. No: ir al paso 11.
   } while (i < n);
   //Paso 11: Fin
}
```

Código 2.43. Imprimir los valores de un vector usando un ciclo for.

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

void main()
{
   int n;//Tamaño del arreglo
   int i;//Índice del arreglo

   //Paso 1: Se define la variable A como arreglo
   int A[10];
   //Paso 2: Se pide el número de datos y se almacena en n
   printf("Tamaño del arreglo: "); scanf_s("%d", &n);
   //Pasos  3,   6  , 5
   for (i = 0; i < n; i++)
   {
        //Paso 4: Se pide el valor de Ai
        printf("A[%d]=", i); scanf_s("%d", &A[i]);
   }
   //Pasos  7,  10  , 9
```

```
for (i = 0; i < n; i++)
   {
        //Paso 8: Se pide el valor de Ai
        printf("A[%d]=%d\n", i, A[i]);
   }
   //Paso 11: Fin
}
```

Código 2.44. Preasignación de valores a un arreglo unidimensional.

```
#include <iostream>

using namespace std;

void main()
{
   //Se declara el arreglo y se asignan valores
   int ar[] = { 1,2,3,4,5,6,7,8,9,10,11};
   //Se declara la variable índice
   int i;

   for (i = 0; i < 11; i++)
        cout << "ar[" << i << "]= " << ar[i] << endl;
}
```

En la preasignación de valores es omisible indicar el número de datos, puesto que el compilador captura automáticamente la cantidad de valores especificados entre llaves. Tal opción es práctica durante la etapa de desarrollo, puesto que ahorra tiempo. Empero, una vez que el programa funcione, se deben incluir los comandos para solicitar información al usuario.

Arreglos bidimensionales

También conocidos como *matrices*, se caracterizan por disponer los datos en tablas de dos dimensiones, como se aprecia en la Figura 2.37. Por ende, se amerita gestionar dos índices en lugar de uno. El Algoritmo 2.16 presenta el procedimiento para asignar valores a una matriz.

Algoritmo 2.16. Definición de los valores de una matriz.

```
1. Pedir el tamaño de la matriz (n,m).
2. For i = 0 hasta n – 1 en incrementos de 1.
3.   For j = 0 hasta m – 1 en incrementos de 1.
```

```
4.      Se pide el valor de A_{i,j}.
5.      Se regresa al paso 3.
6.   Se regresa al paso 2.
7. Imprimir valores de la matriz.
8. Fin.
```

La Figura 2.40 exhibe el diagrama de flujo correspondiente al Algoritmo 2.16, en el que se asignan valores a una matriz y, después, se imprimen. El programa en lenguaje C asociado a esta figura se desglosa en el Código 2.45, de acuerdo con la Figura 2.40. En este código, se predefine un tamaño matricial máximo de 10 × 10. Como consecuencia, los valores de `m` y `n` deben ser menores a 10. Es importante destacar que, si se exceden tales dimensiones, el compilador no generará un error; no obstante, el problema se manifestará durante la ejecución del programa.

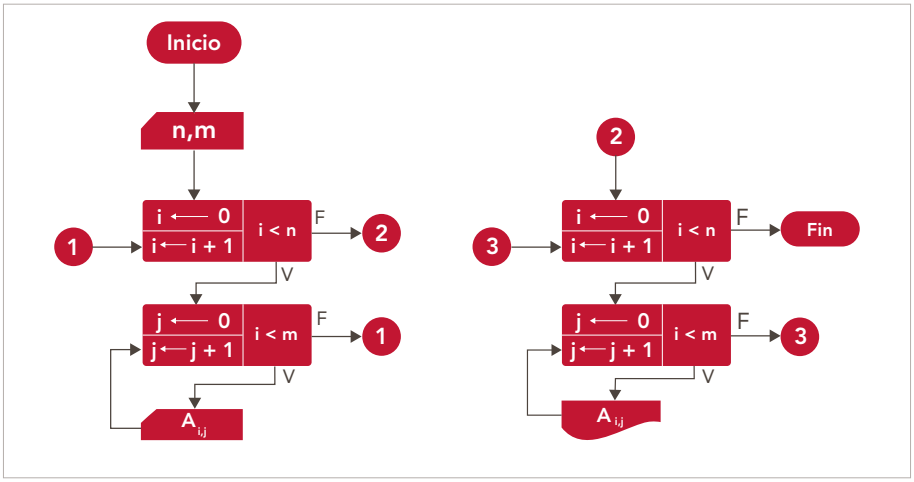


Figura 2.40. Diagrama de flujo para articular una matriz *A* de tamaño *i*, *j*.

Código 2.45. Asignación de valores a una matriz.

```
#include <iostream>

using namespace std;

void main()
{
```

```
    int n, m;
    int A[10][10];
    int i,j;

    //Se piden los valores de n y m
    cout << "Valor de n: "; cin >> n;
    cout << "Valor de m: "; cin >> m;
    // Se asignan los valores a la matriz
    for (i = 0; i < n; i++)
         for (j = 0; j < m; j++)
         {
              cout << "A[" << i << "," << j << "]=";
              cin >> A[i][j];
         }

    //Se imprimen los valores de la matriz.
    cout << "Matriz A =" << endl;
    for (i = 0; i < n; i++)
    {
         for (j = 0; j < m; j++)
              cout << A[i][j] << "\t";
         cout << "\n";
    }
}
```

De un modo análogo, es posible definir de antemano los valores para una matriz, como se observa a continuación.

Código 2.46. Preasignación de valores a una matriz.

```
#include <iostream>

using namespace std;

void main()
{
    int n, m;
    int A[3][4] = { 1,2,3,4,5,6,7,8,9,0,1,2 };
    int i, j;

    //Se piden los valores de n y m.
    n = 3;
    m = 4;
    //Se imprimen los valores de la matriz.
    cout << "Matriz A =" << endl;
    for (i = 0; i < n; i++)
    {
         for (j = 0; j < m; j++)
              cout << A[i][j] << "\t";
         cout << "\n";
    }
}
```

Errores comunes en el manejo de arreglos

Asignar más datos de los reservados

Al declarar un arreglo es necesario indicar la cantidad de elementos que lo conforman. Es posible asignar menos valores, pero no más. De lo contrario, este tipo de error emergerá al momento de ejecutar el programa; el comportamiento dependerá de la plataforma de programación. Se sugiere revisar el Código 2.47 para comprobar lo que ocurre en la situación previamente descrita.

Código 2.47. Asignación de más valores de los declarados.

```
#include <iostream>

using namespace std;

void main()
{
    int ar[10];
    int i;

    for (i = 0; i < 11; i++)
         ar[i] = 10 - i;
    for (i = 0; i < 11; i++)
         cout << "ar[" << i << "]= " << ar[i] << endl;
}
```

Utilizar índices negativos

En particular, al tiempo que los índices varían de forma continua, si alguno de ellos adopta un valor negativo, se generará un error en tiempo de ejecución; por ejemplo, si experimentan cambios continuos durante la operación del programa. Se recomienda observar y poner a prueba el Código 2.48.

Código 2.48. Índices negativos.

```
#include <iostream>

using namespace std;

void main()
{
    int ar[10];
    int i;
```

```
    for (i = -1; i < 10; i++)
         ar[i] = 10 - i;
    for (i = -1; i < 10; i++)
         cout << "ar[" << i << "]= " << ar[i] << endl;
}
```

Cadenas sin carácter nulo

Una cadena de caracteres es un vector compuesto por elementos de tipo **`char`**. Se dispone para almacenar información en formato de texto, lo cual la hace eficaz en tareas que requieran mostrar mensajes, guardar registros textuales o solamente aglomerar caracteres. Algunos ejemplos son: apelativos, direcciones de correo electrónico o etiquetas de archivos. En los lenguajes C y C++, así como en algunos otros, estas cadenas deben finalizar con el carácter nulo (**`\0`**), el cual significa *fin de cadena*. Si este carácter falta, al imprimir se desplegarán secuencias incongruentes o "valores basura". El Código 2.49 presenta un programa que asigna valores predefinidos a un arreglo de tipo **`char`** y, posteriormente, imprime la cadena. La Figura 2.41 ilustra dicho proceso.

Código 2.49. Impresión de una cadena con carácter nulo

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

void main()
{
    char s[22] = { "Programacion Avanzada" };
    int i;
    cout << s;
}
```

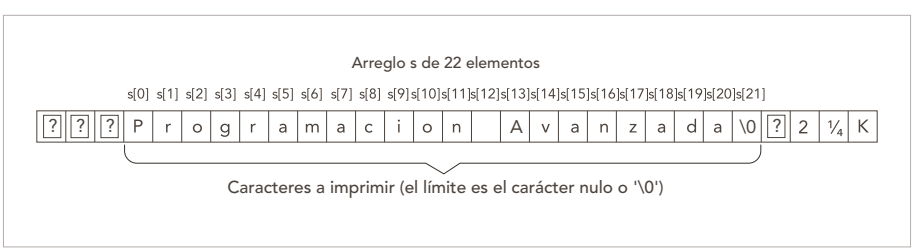


Figura 2.41. Caracteres a imprimir con carácter nulo.

La salida correspondiente al Código 2.49 es:

```
Programacion Avanzada
```

La Figura 2.42 ilustra lo que sucede cuando la variable `s` no incluye el carácter nulo. La impresión de caracteres se prolonga de manera indefinida hasta que el programa encuentra un `\0`. El Código 2.50 muestra lo que ocurre cuando no se marca el final de cadena en ninguna posición de la variable `s`.

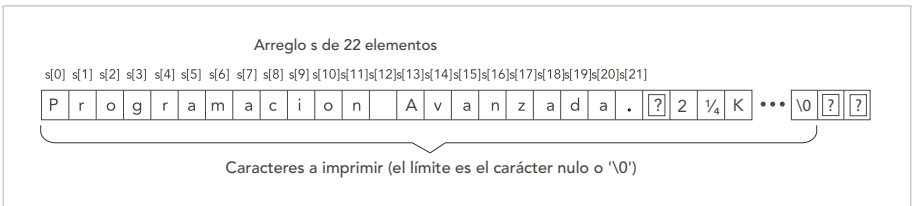


Figura 2.42. Impresión de la variable `s` sin carácter nulo.

Código 2.50. Resultado de la ausencia de carácter nulo.

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

using namespace std;

void main()
{
    char s[22] = { "Programacion Avanzada" };
    int i;

    s[21] = '.'; // Caracter nulo no existe
    cout << s;
}
```

Por consiguiente, la salida contendrá valores basura:

```
Programacion Avanzada. ╠╬╠╬╠╬╠╬¼ºKE² /■
```

Los elementos de una cadena se almacenan en calidad de enteros de 8 *bits*, correspondientes al valor ASCII del carácter que representan. La Tabla 2.18 muestra los valores ASCII de los caracteres más comunes. Solo algunos son imprimibles, aquellos que no lo sean, se representan en las celdas vacías de la tabla.

Tabla 2.18. Relación entre los códigos ASCII y los caracteres.

ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO		ASCII/ SÍMBOLO	
0	\0	26		52	4	78	N	104	h	130	é	156	£	182	Â	208	ð	234	Û
1	☺	27		53	5	79	O	105	i	131	â	157	Ø	183	À	209	Đ	235	Ù
2	☻	28	∟	54	6	80	P	106	j	132	ä	158	×	184	©	210	Ê	236	ý
3	♥	29	↔	55	7	81	Q	107	k	133	à	159	ƒ	185	╣	211	Ë	237	Ý
4	♦	30	▲	56	8	82	R	108	l	134	å	160	á	186	║	212	È	238	¯
5	♣	31	▼	57	9	83	S	109	m	135	ç	161	í	187	╗	213	ı	239	´
6	♠	32	ESC	58	:	84	T	110	n	136	ê	162	ó	188	╝	214	Í	240	
7		33	¡	59	;	85	U	111	o	137	ë	163	ú	189	¢	215	Î	241	±
8		34	“	60	<	86	V	112	p	138	è	164	ñ	190	¥	216	Ï	242	‗
9		35	#	61	=	87	W	113	q	139	ï	165	Ñ	191	┐	217	┘	243	¾
10		36	$	62	>	88	X	114	r	140	î	166	ª	192	└	218	┌	244	¶
11		37	%	63	¿	89	Y	115	s	141	ì	167	º	193	┴	219	▉	245	§
12		38	&	64	@	90	Z	116	t	142	Ä	168	¿	194	┬	220	▄	246	÷
13		39	‘	65	A	91	[	117	u	143	Å	169	®	195	├	221	¦	247	¸
14		40	(	66	B	92	\	118	v	144	É	170	¬	196	─	222	Ì	248	°
15		41	)	67	C	93	]	119	w	145	æ	171	½	197	┼	223	▀	249	¨
16	►	42	*	68	D	94	^	120	x	146	Æ	172	¼	198	ã	224	Ó	250	·
17		43	+	69	E	95	_	121	y	147	ô	173	¡	199	Ã	225	ß	251	¹
18	↕	44	,	70	F	96	`	122	z	148	ö	174	«	200	╚	226	Ô	252	³
19	‼	45	-	71	G	97	a	123	{	149	ò	175	»	201	╔	227	Ò	253	²
20	¶	46	.	72	H	98	b	124	|	150	û	176	░	202	╩	228	õ	254	■
21	§	47	/	73	I	99	c	125	}	151	ù	177	▒	203	╦	229	Õ	255	
22	▬	48	0	74	J	100	d	126	~	152	ÿ	178	▓	204	╠	230	µ		
23	↨	49	1	75	K	101	e	127	⌂	153	Ö	179	|	205	═	231	þ		
24	↑	50	2	76	L	102	f	128	Ç	154	Ü	180	┤	206	╬	232	Þ		
25	↓	51	3	77	M	103	g	129	ü	155	ø	181	Á	207	¤	233	Ú		

El Código 2.51 ejemplifica cómo mostrar en pantalla una parte del alfabeto haciendo uso del código ASCII y el carácter nulo. Se declara un arreglo `s` de tamaño 22 y una variable entera `i`. En el bucle se realiza la suma de 65 a `i` para convertir los valores numéricos en sus correspondientes caracteres ASCII, que representan las letras mayúsculas.

Código 2.51. Asignación de caracteres a una cadena.

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

void main()
{
   char s[22];
   int i;
   for (i = 0; i < 22; i++)
        s[i] = i + 65;
   s[21] = '\0';
   cout << s << " ";
}
```

El resultado que entrega el programa es:

```
ABCDEFGHIJKLMNOPQRSTU
```

2.3.12. APUNTADORES

Una característica sobresaliente de los lenguajes C y C++ son los apuntadores. Dichos elementos guardan la dirección de memoria de otras variables y permiten realizar operaciones directamente en el almacenamiento, acelerando la ejecución y mejorando la eficiencia. Para abordar este tópico, conviene repasar algunos términos técnicos.

Tamaño de variables

Antes de profundizar en el tema de los apuntadores, debe ser considerado que cada variable, dependiendo de su tipo, emplea una cantidad determinada de *bytes* en memoria. El Código 2.52 aborda cómo obtener el tamaño en *bytes* de distintas variables en C y C++.

Código 2.52. Obtención de tamaño de variables.

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

void main()
{
```

```
    int i,n;
    char o;
    float j;
    double k;
    float a[5] = { 0.0, 0.0, 0.0, 0.0, 0.0 };

    i = j = k = 0;
    cout << "Tamaño en bytes de: " << endl;

    n = sizeof(i); cout << "int i = " << n << endl;
    n = sizeof(o); cout << "char o = " << n << endl;
    n = sizeof(j); cout << "float j = " << n << endl;
    n = sizeof(k); cout << "double k = " << n << endl;
    n = sizeof(a); cout << "Arreglo float a de 5 = " << n << endl;
}
```

El resultado es:

```
Tamaño en bytes de:
int i = 4
char o = 1
float j = 4

double k = 8
Arreglo float a de 5 = 20
```

De este modo es como puede obtenerse el tamaño en *bytes* de variables en C y C++.

Dirección de memoria

La dirección de memoria es el lugar físico que el sistema asigna a cada variable del programa. En otras palabras, cuando se declara una variable, el administrador de memoria reserva el espacio necesario acorde al tipo de dato y la disponibilidad del sistema. Esta asignación se lleva a cabo de forma automática y queda fuera del control del programador. La Figura 2.43 ilustra cómo se consignan direcciones de memoria a las distintas variables del Código 2.53 (`int`, `float`, `char`, `double`), mostrando la trayectoria correspondiente a cada una. Es crucial considerar que, al ejecutar el programa, pueden cambiar las direcciones asignadas a las variables. En razón de una gestión de memoria dinámica, el sistema reserva y libera espacio de manera continua.

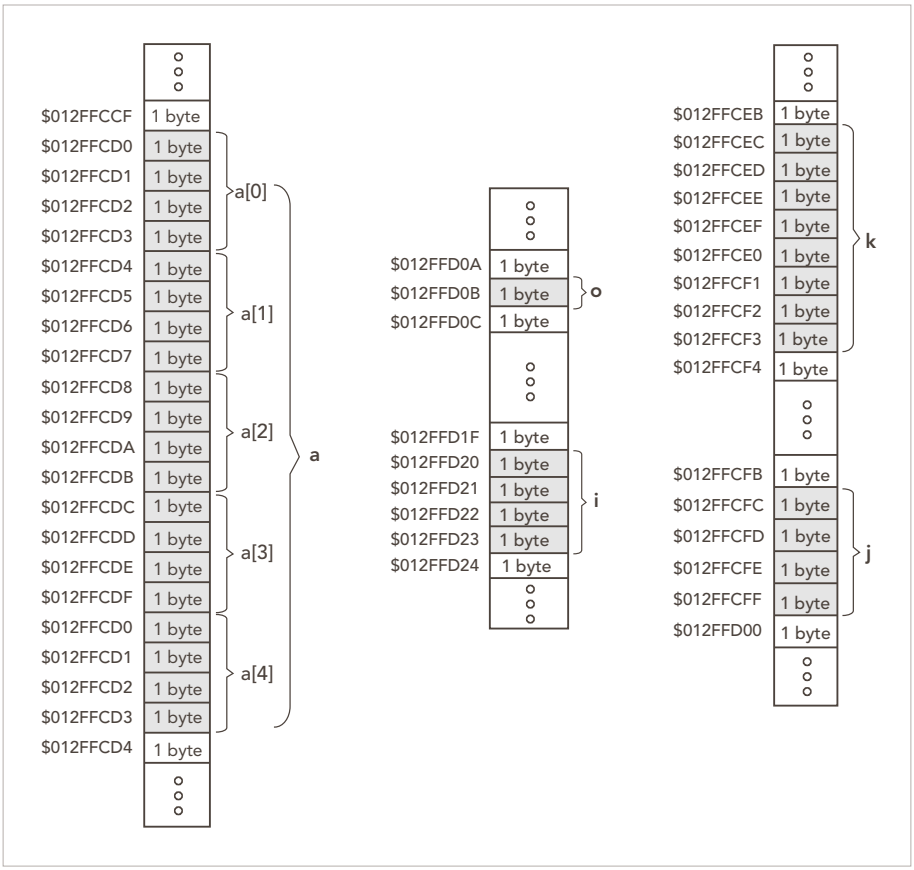


Figura 2.43. Direcciones de memoria de diversas variables.

Código 2.53. Imprimir direcciones de memoria.

```
#include <iostream>
using namespace std;
void main()
{
    int i=0, n;
    char o=1;
    float j=0.0;
    double k=0;
    float a[5] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
```

```
    cout << "Dirección de i = $" << &i << endl;
    printf("Dirección de o = $%p\n", &o);
    cout << "Dirección de j = $" << &j << endl;
    cout << "Dirección de k = $" << &k << endl;
    cout << "Dirección de a = $" << a << endl;
}
```

El resultado de la ejecución será:

```
Dirección de i = $012FFD20
Dirección de o = $012FFD0B
Dirección de j = $012FFCFC
Dirección de k = $012FFCEC
Dirección de a = $012FFCD0
```

Aplicación de los apuntadores

Como se mencionó anteriormente, los apuntadores (también denominados *punteros*) en C y C++ aportan eficiencia a los programas, dado que habilitan la realización de operaciones directamente en memoria. Por tanto, constituyen una herramienta angular en la construcción y manejo de estructuras y registros. Su función consiste en dirigir el flujo a la dirección de memoria de una variable (local o global). La Figura 2.44 ilustra este concepto. La declaración de los apuntadores se ciñe a la siguiente sintaxis:

```
<tipo de dato> *<variable>
```

El tipo de dato puede ser **`int`**, **`float`**, **`char`** o **`double`**, entre otros. El símbolo * indica que se trata de la declaración de un apuntador, mientras que el identificador otorga el nombre a dicha variable. A continuación, se muestran algunos ejemplos:

```
int *p; // Apuntador a un dato de tipo entero (int)
char *o, *u; // Apuntadores a datos de tipo carácter (char)
float *f; // Apuntador a un dato de tipo punto-flotante (float)
```

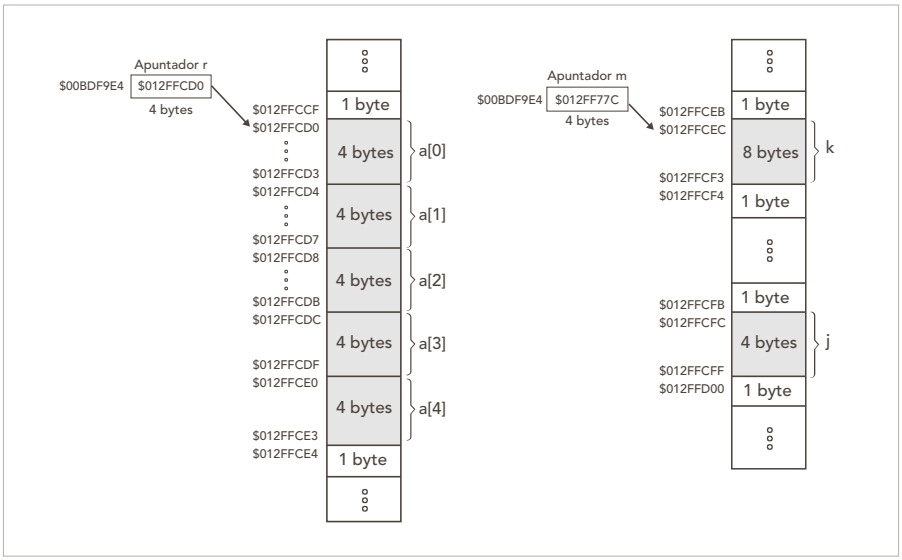


Figura 2.44. Ejemplo de apuntadores a variables.

Los punteros deben inicializarse al asignar la dirección de memoria de la variable a la que se desea "apuntar". La acción se concreta mediante el operador de referenciación `&`, o bien empleando direcciones de memoria previamente almacenadas en otros punteros. Por ejemplo:

```
int i = 16;
int *p, *q;
// Se le asigna al apuntador p la dirección de i
p = &i;
// Se asigna a q la dirección almacenada en p
// que es la dirección de i
q = p;
```

El acceso al valor almacenado en la dirección de un puntero se logra anteponiendo el operador `*` al nombre del apuntador, como se expone en el siguiente ejemplo:

```
int i = 231;
int j, *p;
p = &i;
// Imprime el valor de i
cout << "i =  " << *p << endl;
j = *p - 100; // Se asigna el valor de i - 100
```

Operaciones con apuntadores

Los procesos más habituales con apuntadores corresponden a las operaciones aritméticas de suma y resta, las cuales permiten desplazarse a lo largo de un bloque de memoria, por lo general un arreglo de datos. La Figura 2.45 ilustra el efecto de incrementar o decrementar el valor del apuntador. Cabe aclarar que dichas operaciones solo tienen impacto en la dirección designada.

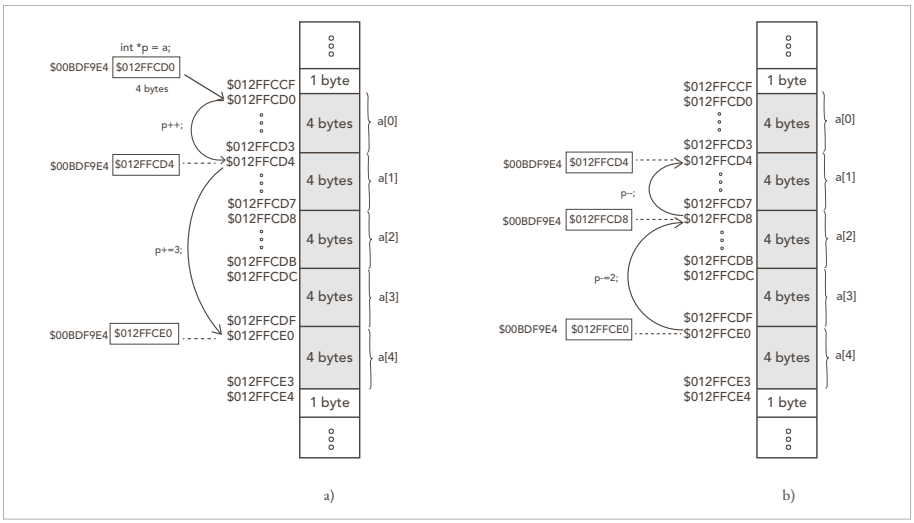


Figura 2.45. Operaciones en el apuntador. (a) Incremento. (b) Decremento.

Es necesario tener cautela al realizar operaciones de incremento o decremento en apuntadores, puesto que podrían exceder el rango de memoria asignado. En el peor de los escenarios, pueden generarse errores en tiempo de ejecución. El Código 2.54 ejemplifica la apliacación de un apuntador para asignar valores a un vector.

Código 2.54. Asignación de valores de un arreglo con apuntadores.

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

void main()
{
   int *p;       //Se declara el apuntador.
                 // Se declara un vector a, del mismo tipo que el apuntador.
```

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

    p = a;
    for (i = 0; i < 10; i++)
        *p++ = i;
    for (i = 0; i < 10; i++)
        cout << a[i] << endl;
}
```

Asimismo, el apuntador puede emplearse para imprimir los valores de un arreglo, como se muestra en el Código 2.55.

Código 2.55. Uso de apuntadores para imprimir los valores de un arreglo.

```
#include <iostream>
using namespace std;
void main()
{
    int *p;       // Se declara el apuntador.
                  // Se declara un vector a, del mismo tipo que el apuntador.
    int a[10];
    int i;
    p = a;
    for (i = 0; i < 10; i++)
        *p++ = i;
    for (i = 0; i < 10; i++)
        cout << *p-- << endl;
}
```

Uso de apuntadores con funciones

Los apuntadores pueden emplearse para intercambiar información entre funciones. En el Código 2.56, la función `cambiar_dato()` modifica los valores de las variables locales `m` y `n` del bloque `main()` a través de apuntadores.

Código 2.56. Uso de apuntadores en funciones.

```
void cambiar_dato(int *a, double *b);

void cambiar_dato(int *a, double *b)
{
```

```
    *a = 10;
    *b = 35.675;
}

void main()
{
    int *p,m;// Se declara el apuntador p
    double *q,n;// Se declara el apuntador q

    m = 12;
    n = 0.5;
    cout << "Valores iniciales\n";
    cout << "m = " << m << "\t" << " n = " << n << endl;
    p = &m;
    q = &n;
    cout << "Se manda cambiar los valores\n";
    cambiar_dato(p, q);
    cout << "m = " << m << "\t" << " n = " << n << endl;
}
```

Ejercicio: Usar apuntadores para convertir un valor decimal a binario

Comprensión del problema

Lo primero que debe comprenderse es el proceso de conversión de un número decimal a uno binario. Para ejemplificarlo, se convertirá el valor decimal 931 a binario, como se muestra en la Figura 2.46. La conversión se realiza dividiendo la cifra original sucesivamente entre dos, hasta obtener un cociente igual a cero, o bien, completar el número de *bits* asignados para la representación.

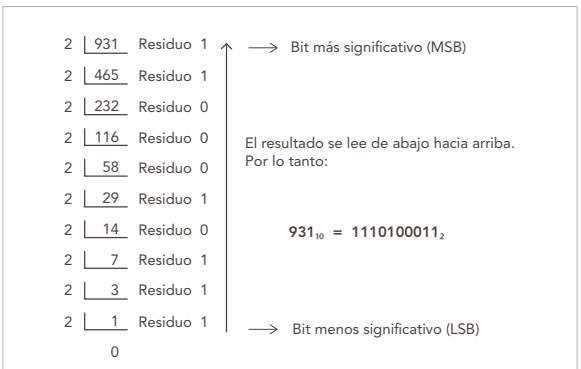


Figura 2.46. Ejemplo de conversión de decimal a binario.

Algoritmo

Con base en la Figura 2.46, se propone el diseño del Algoritmo 2.17, orientado a plantear la solución al problema.

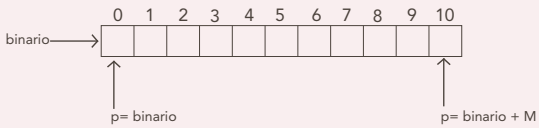
Algoritmo 2.17. Conversión de decimal a binario.

```
1. Se pide el valor N a convertir.
2. Se pide el número M de bits.
3. Se inicia el contador i en 0.
4. Se obtiene el residuo (módulo) de N/2.
5. El residuo se almacena en la cadena.
6. Se asigna a N el resultado entero de N/2.
7. Se incrementa i.
8. ¿i es menor que M? Sí: Ir al paso 4. No: Ir al paso 9.
9. Se imprime la cadena resultante.
10. Fin.
```

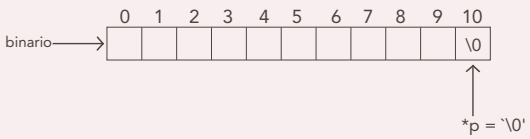
Es necesario, si se desea utilizar apuntadores, definir la manera como se almacenará el resultado en la cadena. Para ello, se plantean los pasos del Algoritmo 2.18.

Algoritmo 2.18. Manejo del apuntador para la conversión de decimal a binario.

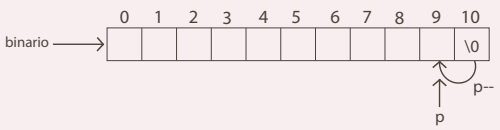
```
1. Se decide usar una cadena "binario", de tamaño M.
2. Se usa un apuntador tipo char p.
3. El apuntador p se inicializa apuntando a binario + M. Esto debido a que el
   primer resultado se coloca en el extremo derecho, y el segundo en el extremo
   izquierdo.
```




```
4. Se asigna el carácter nulo al contenido de la dirección de p.
```




```
5. Se decrementa la dirección en 1.
6. Se inicializa la variable i en 0.
```



```
7. Se asigna al contenido de la dirección p el módulo 2 de N. Se le suma 48 para
   que corresponda al carácter 0 o 1.
8. Se decrementa la dirección de p.
9. Se divide el valor de N entre 2.
10. Se incrementa i en 1.
11. ¿Es i menor que M? Sí: Ir al paso 7. No: Ir al paso 12.
12. Se asigna al contenido de la dirección p el módulo 2 de N. Se le suma 48 para
    que corresponda al carácter 0 o 1.
13. Se imprime el binario.
14. Fin.
```

Diagrama de flujo

En función del Algoritmo 2.17, se traza el diagrama de flujo que facilita su implementación en lenguaje C, mostrado en la Figura 2.47. Para representar la secuencia repetitiva de operaciones, se emplea un ciclo **`for`**, iniciado en 0 y que permanece activo mientras el valor de **`i`** sea menor que **`M`**.

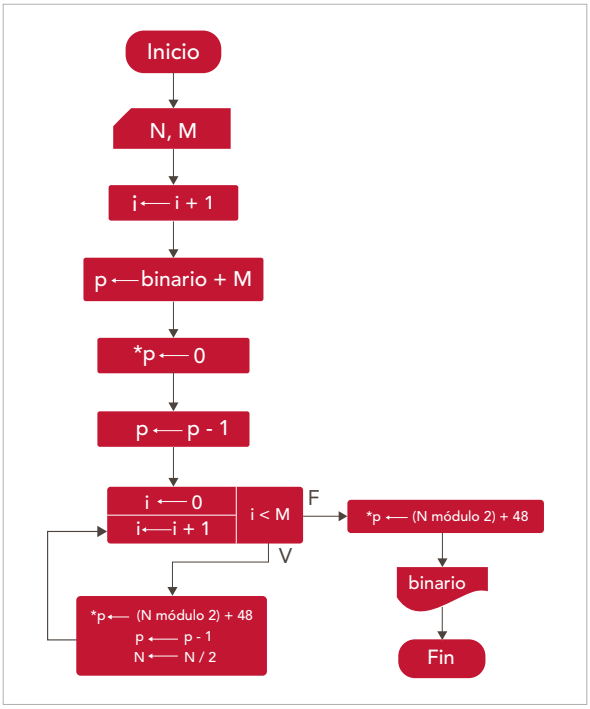


Figura 2.47. Diagrama de flujo para la conversión de decimal a binario.

Código en C

El Código 2.57 se ejerce para convertir un valor decimal a su representación binaria, almacenando el resultado en una cadena de caracteres.

Código 2.57. Conversión de decimal a binario.

```
#include <iostream>
#include <conio.h>
using namespace std;

void main()
{
   char binario[12];
   char* p;
   int i, N, M;

   cout << "Convertidor decimal a binario " << endl;
   N = 931; // Valor a convertir
   M = 11; // Numero de bits

   p = binario + M;
   *p-- = 0;
   for (i = 1; i < M; i++)
   {
        *p = (N % 2) + 48;
        p--;
        N /= 2;
   }
   *p = (N % 2) + 48;
   cout << binario << endl;
   _getch();
}
```

El resultado de la ejecución es:

```
Convertidor decimal a binario
01110100011
```

2.3.13. MEMORIA DINÁMICA EN C++

Es habitual que sea imposible anticipar la cantidad de variables necesarias en un programa. Un ejemplo sencillo es el ordenamiento de números: al desarrollar el código, con frecuencia

se desconoce la cantidad exacta de datos que se procesarán, por lo que se habitúa a declarar arreglos de un tamaño mayor al necesario para asegurar memoria suficiente. En el pasado, al declarar un arreglo, su tamaño permanecía inamovible durante el tiempo de ejecución, lo que se conoce como *asignación estática*. En cierta medida, al efectuar apuntadores y funciones en C, es admisible realizar asignaciones de memoria en tiempo de ejecución, es decir, de forma *dinámica*. El lenguaje C dispone de cuatro funciones primordiales para la gestión flexible de memoria: **`malloc()`**, **`calloc()`**, **`realloc()`** y **`free()`**. Además, en C++ actúan los operadores **`new`** y **`delete`**. En contraste con las variables declaradas de manera estática, cuyo espacio de memoria se definen en la compilación, las variables dinámicas reservan memoria en el momento de la ejecución. Dicho mecanismo puede aplicarse a cualquier tipo de variable, estructura o clase, siempre que se gestionen correctamente tales funciones.

Función `malloc()`

Tal función reserva un bloque específico de memoria en *bytes* y devuelve a un apuntador la dirección de inicio de dicho bloque. El prototipo de la función es el siguiente:

```
void* = malloc(tamaño en bytes);
```

Una vez reservada, la memoria puede aprovecharse como un arreglo de datos. En el Código 2.58 se asigna espacio para 10 datos de tipo entero y posteriormente se imprimen sus valores empleando el formato de un arreglo.

Código 2.58. Uso de la función `malloc()`.

```
#include <iostream>

using namespace std;

void main(void)
{
    int* c;
    int i;

    //Uso de malloc para generar variables sencillas
    cout << "Reservando 10 localidades de memoria para datos int\n";

    c = (int*) malloc(10 * sizeof(int));

    for (i = 0; i < 10; i++)
    {
```

```
        cout << c[i] << "\t";
    }

    free(c);
}
```

El resultado corresponde al valor predeterminado que contiene el espacio de memoria en el momento de su creación.

```
Reservando 10 localidades de memoria para datos int

-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
```

Función `calloc()`

El comportamiento de la función **`calloc()`** es similar al de **`malloc()`**, con la particularidad de que inicializa a cero todos los *bytes* de la memoria reservada. A continuación se muestra el prototipo de la función:

```
void* = calloc(Cantidad, tamaño en bytes);
```

Donde el primer argumento, **`Cantidad`**, denota el número de elementos requeridos, mientras que el segundo indica el tamaño en *bytes* correspondiente al tipo de dato.

Código 2.59. Uso de la función `calloc()`.

```
#include <iostream>

using namespace std;

void main(void)
{
```

```
    int* c;
    int i;

    //Uso de malloc para generar variables sencillas
    cout << "Reservando 10 localidades de memoria para datos int\n";
    c = (int*)calloc(10, sizeof(int));
    for (i = 0; i < 10; i++)
    {
        cout << c[i] << "\t";
    }
    free(c);
}
```

El resultado es el siguiente:

```
Reservando 10 localidades de memoria para datos int

0
0
0
0
0
0
0
0
0
0
```

En este caso, las 10 localidades se inicializaron con el valor 0. La Tabla 2.19 presenta los valores asignados por defecto al ejecutar la función **`calloc()`** en diferentes tipos de variables, tomando como base los Códigos 2.58 y 2.59.

Tabla 2.19. Resultados de usar `malloc()` y `calloc()` para diferentes tipos de variables.

INT		FLOAT		DOUBLE		CHAR	
malloc	calloc	malloc	calloc	malloc	calloc	malloc	calloc
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL
-842150451	0	-4.31602e+08	0	-6.27744e+66	0	=	NULL

Función `realloc()`

La función redimensiona un bloque de memoria previamente asignado. Si la memoria se amplía, se conservan los valores existentes y se agrega espacio adicional. De manera contraria, cuando la memoria se reduce, la información sobrante es eliminada por el sistema al reasignar ese espacio a otro proceso en ejecución. El Código 2.60 muestra el uso de la función **`realloc()`**. El primer paso consiste en asignar memoria al puntero, para después almacenar valores e imprimirlos. Luego, la memoria se redimensiona a un tamaño de 15 y los valores se despliegan nuevamente. Al final, el tamaño se reduce a 5 y se realiza una última impresión.

Código 2.60. Uso de la función `realloc()` para datos tipo entero.

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

void main(void) {
    int* c;
    int i;

    cout << "Reservando 10 localidades de memoria para datos int\n";
    c = (int*)calloc(10, sizeof(int));
    for (i = 0; i < 10; i++)
    {
         c[i] = i;
         cout << c[i] << "\t";
    }

    cout << "\nSe redimensiona a 15 datos\n";
    c = (int*)realloc(c, 15 * sizeof(int));
    for (i = 0; i < 15; i++)
    {
         cout << c[i] << "\t";
    }

    cout << "\nSe redimensiona a 5 datos\n";
    c = (int*)realloc(c, 5 * sizeof(int));
    for (i = 0; i < 5; i++)
    {
         cout << c[i] << "\t";
    }
    free(c);
}
```

El resultado es el siguiente:

```
Reservando 10 localidades de memoria para datos int
0       1       2       3       4       5       6       7       8       9
```

```
Se redimensiona a 15 datos
0       1       2       3       4       5       6       7       8       9
-842150451      -842150451      -842150451      -842150451      -842150451

Se redimensiona a 5 datos
0       1       2       3       4
```

Función `free()`

Se ejecuta para liberar la memoria reservada a través de los métodos **`malloc()`** y **`calloc()`**. Tal función es imprescindible, ya que, de no emplearse, la memoria permanece bloqueada, inutilizable hasta que el sistema se reinicie. Los Códigos 2.58, 2.59 y 2.60 ilustran el empleo de esta función.

`new` y `delete`

Son comandos ampliamente usados en la programación orientada a objetos, aunque también están disponibles en C. Su funcionamiento es similar al de **`malloc()`** y **`free()`**, la diferencia radica en que no permiten redimensionar la memoria. En el Código 2.61 se reserva memoria en tiempo de ejecución para un arreglo tipo **`float`**. La variable `n` indica el número de localidades de memoria a reservar. Por el contrario en **`malloc()`**, no es requisito conocer el tamaño en *bytes* de la variable. El comando **`new`** guarda la memoria, mientras que **`delete`** destruye la variable y libera el espacio.

Código 2.61. Uso de los comandos `new` y `delete` para el manejo de memoria dinámica.

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

using namespace std;

void main()
{
    int n = 15;
    float* a = new float[n];

    for (int i = 0; i < n; i++)
         a[i] = i * 0.3389;

    for (int i = 0; i < n; i++)
         cout << a[i] << '\t';
```

```
    delete a;
    _getch();
}
```

Ejercicio: Uso de arreglos con memoria dinámica

La memoria dinámica puede ser también utilizada para generar arreglos bidimensionales. A continuación, se llevará a cabo el proceso correspondiente.

Comprensión del problema

En el enfoque más tradicional, el tamaño de un arreglo bidimensional (o matriz) se define al momento de crear la variable y es inamovible durante la ejecución del programa. En cambio, mediante la memoria dinámica es posible modificar el tamaño del arreglo durante la ejecución, lo que permite ajustarlo según las necesidades y evitar el desperdicio de memoria.

Algoritmo

En este caso, el Algoritmo 2.19 muestra los pasos a seguir para la construcción del arreglo dinámico; se definen valores para cada elemento, y finalmente, se manda imprimir los valores para verificar que se asignaron adecuadamente. El Código 2.62 expone el programa para generar una matriz usando memoria dinámica.

Algoritmo 2.19. Manejo del apuntador para la conversión de decimal a binario.

```
1. Se pide el número de renglones (nR) y de columnas (nC).
2. Se reserva memoria para A según el número de renglones.
3. Se reserva memoria para las columnas de cada renglón de A.
```

Código en C

Código 2.62. Programa para generar un arreglo usando apuntadores.

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

```
int** A;
int nR, nC;

void main()
{
    int i, j;

    cout << "Número de renglones: "; cin >> nR;
    cout << "Número de columnas : "; cin >> nC;
    //Se reserva memoria para R renglones de la matriz.
    //Se reserva para variables tipo apuntador-entero.
    A = (int**)malloc(nR * sizeof(int*));
    //Para asignar el tamaño de cada una de las filas o arreglos
    //unidimensionales
    for (i = 0; i < nR; i++) {
         A[i] = (int*)malloc(nC * sizeof(int));
    }
    // Se asignan valores al arreglo
    for (int k = 0, i = 0; i < nR; i++)
         for (j = 0; j < nC; j++, k++)
              A[i][j] = k;
    // Se imprimen los valores
    for (int k = 0, i = 0; i < nR; i++)
    {
         for (j = 0; j < nC; j++, k++)
              cout << A[i][j] << "\t";
         cout << endl;
    }
    // Se libera la memoria
    for (int i = 0; i < nR; i++) {
         free(A[i]);
    }
    free(A);
}
```

El resultado de este código se muestra a continuación:

```
Número de renglones: 5
Número de columnas: 6
0               1       2       3       4       5
6               7       8       9       10      11
12              13      14      15      16      17
18              19      20      21      22      23
24              25      26      27      28      29
```

También se puede usar el comando **new** para crear el arreglo matricial de dos dimensiones. El proceso se muestra en el Código 2.63.

Código 2.63. Arreglo bidimensional o matriz usando el comando new.

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

```
int** A;
int nR, nC;

void main()
{
    int i, j;

    //Se pide la dimensión de la matriz
    cout << "Número de renglones: "; cin >> nR;
    cout << "Número de columnas : "; cin >> nC;
    //Se reserva memoria para R renglones de la matriz.
    //Se reserva para variables tipo apuntador-entero.
    A = new int*[nR];

    //Para asignar el tamaño de cada una de las filas o arreglos
    //unidimensionales
    for (i = 0; i < nR; i++) {
         A[i] = new int[nC];
    }

    // Se asignan valores al arreglo
    for (int k = 0, i = 0; i < nR; i++)
         for (j = 0; j < nC; j++, k++)
              A[i][j] = k;

// Se imprimen los valores
    for (int k = 0, i = 0; i < nR; i++)
    {
         for (j = 0; j < nC; j++, k++)
              cout << A[i][j] << "\t";
         cout << endl;
    }
    // Se libera la memoria
    for (int i = 0; i < nR; i++) {
         delete A[i];
    }
    delete A;
}
```

2.3.14. MANEJO DE EXCEPCIONES

Las excepciones son errores producidos en tiempo de ejecución, es decir, mientras el programa se efectúa, razón por la cual el compilador difícilmente puede anticiparlos. Este tipo de errores puede originarse al ejecutar operaciones matemáticas, asignaciones de memoria o escrituras de datos fuera de un espacio reservado —como sucede con los arreglos—, entre más situaciones de la misma envergadura.

Una forma de prevención consiste en implementar bloques **`try-catch()`**, diseñados con el fin de "atrapar" excepciones en tiempo de ejecución. Tales bloques contribuyen a evitar la interrupción o el cierre inesperado del programa mientras corre. A continuación se muestra la sintaxis respectiva.

```
try
{
    //Bloque de instrucciones 1
    throw tipo de error;
    //Bloque de instrucciones 2
}
catch (tipo variable)
{
    //Bloque de instrucciones 3
}
```

El **Bloque de instrucciones 1** se ejecuta antes de generar el llamado a la excepción a través del comando **throw**. Una vez lanzada, se envía un código asociado al tipo de error, el cual puede adoptar distintos valores para determinar variables, por ejemplo: **int**, **float**, **double**, entre otros. El valor enviado por **throw** es capturado en **catch()** mediante una variable del mismo tipo que el generado en **throw**. El **Bloque de instrucciones 3** se ejecuta únicamente cuando se produce la excepción, de modo que el **Bloque de instrucciones 2**, localizado después de **throw**, no llega a ejecutarse. A continuación, algunos ejemplos.

Bloque try-catch() básico

Ante el cálculo de la raíz cuadrada de una variable, se presenta un problema cuando el valor del radicando es negativo. La Figura 2.48 diagrama los flujos correspondientes al problema tanto sin como con el bloque **try-catch()**. En lo que atañe al Código 2.63, se condensa el diagrama de la Figura 2.48(a), mientras que el de la Figura 2.48(b) refiere al Código 2.64.

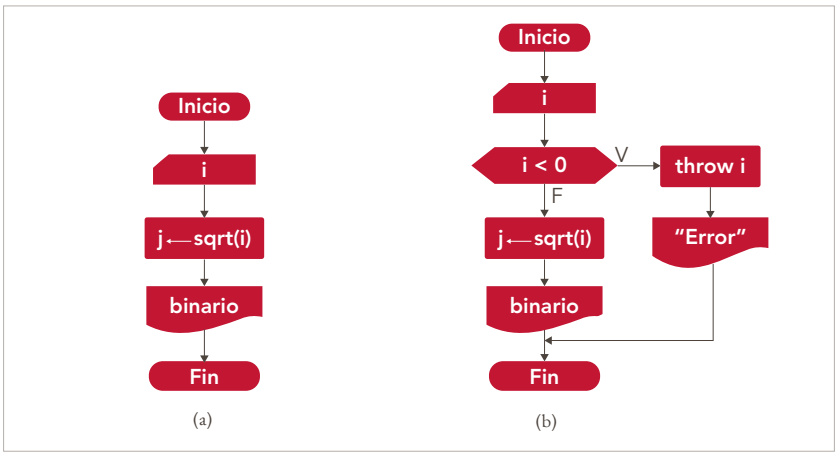


Figura 2.48. Cálculo de la raíz cuadrada de una variable. (a) Sin `try-catch()`. (b) Con `try-catch()`.

Código 2.64. Calcular la raíz cuadrada.

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

void main()
{
    int i;
    float j;

    i = -2;
    j = (float) sqrt(i);
    cout << j;
}
```

Se obtiene el siguiente resultado:

```
-nan(ind)
```

El Código 2.64 previene este tipo de errores mediante la ejecución del bloque **`try-catch()`**.

Código 2.64. Control de errores en el cálculo de la raíz cuadrada a través de `try-catch()`.

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

void main()
{
    int i;
    float j;

    i = -2;
    j = 0;
    try
    {
         if (i < 0) throw 1;
         j = (float)sqrt(i);
         cout << j;
    }
    catch (int e)
    {
         cout << "No se puede obtener la raíz cuadrada de un valor negativo" << endl;
    }
}
```

En este caso, se produce el siguiente resultado:

```
No se puede obtener la raíz cuadrada de un valor negativo
```

El dato administrado por el comando **throw** puede codificarse en distintos tipos. Por ejemplo, el Código 2.64 puede alterarse para que dicha instrucción envíe un valor tipo **double**; esta variación resulta en el Código 2.65.

Código 2.65. Uso de throw tipo double.

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

void main()
{
    int i;
    float j;

    i = -2;
    j = 0;
    try
    {
         if (i < 0) throw 1.0;
         j = (float)sqrt(i);
         cout << j;
    }
    catch (double e)
    {
         cout << "No se puede obtener la raíz cuadrada de un valor negativo" << endl;
    }
}
```

Asimismo, **throw** puede enviar el valor de una variable, como se muestra en el Código 2.66. Es importante subrayar que, si la variable **i** difiere en tipo con **e**, el manejo de la excepción será inefectivo, por lo que el comando **j = (float)sqrt(i)** desencadenará el cierre abrupto del programa.

Código 2.66. Envío del valor de una variable usando el comando throw.

```
#include <iostream>

using namespace std;

void main()
{
    int i;
    float j;

    i = -2;
    j = 0;
    try
    {
         if (i < 0)
              throw i;
```

```
        j = (float)sqrt(i);
        cout << j;
    }
    catch (int e)
    {
        cout << "Error: No se puede obtener la raíz cuadrada de " << i << endl;
    }
}
```

El Código 2.67 muestra cómo generar diferentes tipos de **`throw`**, lo que permite al usuario gestionar diversas excepciones según las necesidades del programa. Se capturan los errores de tipo **`int`**, **`float`** y de cadena; las incidencias de tipo **`char`** se incluyen en la última opción, encargada de manejar las alteraciones no definidas previamente.

Código 2.67. Uso de diferentes tipos de excepciones.

```
#include <iostream>

using namespace std;

void main()
{
    int i;
    float j;

    i = 2;
    j = 0.0;
    try
    {
        if (i < 0) throw i;
        if (j > 10.0) throw j;
        if (i == 0 && j == 0) throw "Error";
        if (j == 0) throw 'o';
        cout << "Todo bien";
    }
    catch (int e)
    {
        cout << "Se recibió el código entero " << e << endl;
    }
    catch (float e)
    {
        cout << "Se recibió el código decimal " << e << endl;
    }
    catch (char* e)
    {
        cout << "Se recibió el código " << e << endl;
    }
    catch (...)
    {
        cout << "Se recibió un código desconocido" << endl;
    }
}
```

De igual forma, las excepciones pueden generarse dentro de funciones, como se muestra en el Código 2.68, donde calcular la raíz cuadrada de un valor negativo suscita una excepción de tipo `int`. La salida de este Código solo imprime el primer resultado, después se indica el error y el resto de los cálculos dejan de realizarse.

Código 2.68. Captura de excepciones desde funciones.

```
#include <iostream>

using namespace std;

float raiz(float a)
{
    if (a < 0) throw a;
    return sqrt(a);
}

void main()
{
    float j, k;

    j = 5;
    k = 0;
    try
    {
         k = raiz(j);
         cout << "k = " << k << endl;
         k = raiz(-4);
         cout << "k = " << k << endl;
         k = raiz(8);
         cout << "k = " << k << endl;
    }
    catch (...)
    {
         cout << "No se puede calcular raíz a un valor negativo." << endl;
    }
}
```

El Código 2.69 implementa la captura directa de errores en las funciones. Si el argumento es positivo, se calcula la raíz; de lo contrario, se indica la excepción, lo cual evita que esta se produzca en tiempo de ejecución y que el programa se cierre abruptamente.

Código 2.69. Captura de excepciones directamente en la función.

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

float raiz(float a)
{
    try
    {
```

```
        if (a < 0) throw a;
        return sqrt(a);
    }
    catch (...)
    {
        cout << "No se puede calcular raíz a un valor negativo." << endl;
    }
}

void main()
{
    cout << raiz(4.0) << endl;
    cout << raiz(5.5) << endl;
    cout << raiz(-7) << endl;
    cout << raiz(8) << endl;
}
```

2.3.15. SOBRECARGA DE FUNCIONES

Tal capacidad permite atribuirle a una función más de una forma, de modo que el compilador, según la cantidad de argumentos o sus tipos, seleccione la definición adecuada. Ahora bien, C y C++ incluyen vastas opciones de sobrecarga; por ejemplo, **`pow(x,y)`**, por defecto en la biblioteca de funciones matemáticas **`cmath`**, se encuentra sobrecargada en C++ para invocar una u otra definición, dependiendo del tipo de datos de los argumentos. Por ende, estas llamadas se tratarán de manera independiente. Cabe señalar que la sobrecarga de funciones constituye más una ventaja del lenguaje que un planteamiento algorítmico, motivo por el cual el desarrollo de este último se omitirá en la solución de los ejercicios.

Ejercicio: Usar sobrecarga de funciones para calcular áreas

Para sobrecargar una función, es necesario incorporar diferentes tipos o cantidades de argumentos. En el Código 2.70 se calcula el área de una figura geométrica (cuadrado, triángulo o círculo) usando diferentes combinaciones de argumentos en la función **`Area()`**.

Código 2.70. Cálculo del área con sobrecarga de funciones.

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

#define pi 3.14159
```

```
float Area(float r);
int Area(int a);
float Area(float b, float h);

void main()
{
    float c;
    int d;

    c = 15.5;
    d = 13;

    printf("Área del círculo = %.2f\n", Area(c));
    printf("Área del cuadrado = %d\n", Area(d));
    printf("Área del triángulo = %.2f\n", Area(c, (float)d));
    _getch();
}

//Área de un círculo
float Area(float r)
{
    return 2 * pi * r * r;
}

//Área de un cuadrado
int Area(int a)
{
    return a * a;
}

//Área de un triángulo
float Area(float b, float h)
{
    return b * h / 2;
}
```

Ejercicio: Usar sobrecarga de la función `Potencia()`

En este ejercicio se desarrolla una función con la capacidad de elevar un número, ya sea de categoría `int` o `float`, a una potencia entera. Para ello, se emplea el Código 2.71.

Código 2.71. Sobrecarga de la función `Potencia()`.

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

using namespace std;
```

```
int Potencia(int n, int p);
float Potencia(float n, int p);

int Potencia(int n, int p)
{
    int R = 1, i;

    for (i = 1; i ≤ p; i++)
         R *= n;
    return R;
}

float Potencia(float n, int p)
{
    int i;
    float R = 1.0;

    for (i = 1; i ≤ p; i++)
         R *= (float)n;
    return R;
}

void main()
{
    int i, j, k;
    float l, n;

    i = 3; j = 5;

    k = Potencia(i, j);
    cout << i << "^" << j << "=" << k << endl;

    n = 1.5; j = 4;
    l = Potencia(n, j);
    cout << n << "^" << j << "=" << l << endl;

    _getch();
}
```

2.3.16. VARIABLES TIPO `static`

En principio, el tipo **`static`** se aplica a las variables locales; el aspecto clave es que estas conserven su valor previo al iniciar nuevas ejecuciones. Es decir, cuando una función termina su ejecución, todas las locales se destruyen y la memoria reservada se libera. Ahora bien, al anteponer la palabra reservada **`static`** a la designación de una variable, esta permanece intacta y conserva su valor. Así, al invocar nuevamente la función, la variable retoma su valor previo. El proceso se ilustra en el Código 2.72.

Código 2.72. Ejemplo del uso de variables estáticas.

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

//función local con la variable estática tipo entero j
int Prueba(int k)
{
    static int j = 1;       //Se asigna el valor inicial
    j *= k;                                 //El valor j no se pierde aun
    return j;                         //cuando la función se termina
}

void main()
{
    //En la primera ejecución, j vale 1
    cout << Prueba(3) << endl;//Se retorna j = 3
    cout << Prueba(5) << endl;//Se retorna j = 15
    cout << Prueba(2) << endl;//Se retorna j = 30
}
```

2.4. EJERCICIOS DE REPASO

En los siguientes ejercicios propuestos, es imperativo ceñirse la metodología descrita en la Figura 1.1. Se debe escribir el algoritmo, elaborar el diagrama de flujo y desarrollar el código del programa. Para prevenir errores en tiempo de ejecución, es necesario aplicar los métodos de captura de excepciones revisados en el capítulo.

1. Convertir un valor fraccionario de decimal a binario. Se debe especificar el número de *bits* de la parte entera y el número de *bits* de la parte fraccionaria.

2. Desarrollar un programa que complete la siguiente tabla de conversiones:

VALOR	BINARIO	OCTAL	HEXADECIMAL	ERROR DE CONVERSIÓN
129	10000001	201		
-1		777	1FF	
-115	110001101	615		
48693	1011111000110101		BE35	
15896477		74507452	F28F2A	
3.14159				
24.2658				
-1.6309				
25.690				

3. Realizar un programa que complete la siguiente tabla de conversiones, respetando las limitaciones de *bits*:

VALOR	BINARIO			EQUIVALENTE		ERROR***
	Bits sin límite	Con *N bits*		Hex*	Oct**	
129	10000001	*N* = 5	00001			
-1	111111111	*N* = 7	1111111			
-115	110001101	*N* = 8	10001101			
48693	1011111000110101	*N* = 8	00110101			
15896477	111100101000111100101010	*N* = 16	1000111110011101			
3.14159		*N* = 32				
24.2658		*N* = 32				
-1.6309		*N* = 64				
25.690		*N* = 64				

* Hexadecimal, ** Octal, *** Error de conversión.

4. Verificar y corregir el siguiente código para determinar si un número es par o impar.

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

void main()
{
    int N;
    int R;

    N = 35;        //Paso 1: Conocer el valor de N
    R = N / 2;     //Paso 2: Calcular el módulo 2
    if (R)         //Paso 3: Preguntar por el resultado
         printf("Es impar");          //Paso 4
    else
         printf("Es par");            //Paso 6
    //Paso 7
    _getch();
}
```

5. Determinar si un valor flotante es entero. Por ejemplo, 1.5325 no es entero, mientras que 2.0000 sí lo es.

6. Identificar si un número es primo.

7. Imprimir el triángulo de Pascal hasta un nivel *N*.

8. Realizar la combinación de dos números.

9. Calcular la permutación de dos números.

10. Encontrar una raíz de una función usando el método del punto medio.

11. Obtener el promedio, la desviación estándar y la moda de una lista de *n* datos.

12. Ejecutar las operaciones de suma, resta, multiplicación y división de números complejos para datos de tipo entero, flotante y doble. Utilizar sobrecarga de funciones.

13. Leer una cadena de caracteres y separarla en palabras, delimitadas por espacios. Por ejemplo, la cadena "Hola a todos. Espero que estén bien.", se debe separar en las palabras *Hola*, *a*, *todos*, *Espero*, *que*, *estén, bien*. Cada palabra se denominará *token*.

14. Almacenar cada *token* del ejercicio 13 en un arreglo de cadenas.

15. Resolver los siguientes ejercicios empleando memoria dinámica y apuntadores.

    a) Ordenar N números de mayor a menor.

    b) Calcular la suma de dos matrices de tamaño $N \times M$.

    c) Realizar la multiplicación de dos matrices de tamaño $N \times M$.

    d) Resolver un sistema de N ecuaciones con N incógnitas recurriendo al método de Gauss-Jordan.

    e) Calcular un determinante de tamaño máximo 10. Usar sobrecarga de funciones.

    f) Obtener la inversa de una matriz $N \times N$.

Programación orientada a objetos

3. Programación orientada a objetos

3.1. ESTRUCTURAS

Las estructuras en lenguaje C/C++, también denominadas *registros* (**record**), constituyen una herramienta sustancial para el desarrollo de programas, ya que permiten agrupar diferentes tipos de datos dentro de una variable y, por lo general, son usadas con un próposito específico. Hasta este punto solo se han descrito variables que pueden almacenar un único tipo de dato; es decir, las de tipo **int** permiten solo valores enteros, y los arreglos siguen la misma lógica. Pero los registros facilitan mantener un control organizado sobre la información, lo cual sería imposible con un arreglo o variable individual.

La sintaxis para declarar una estructura se expone en el Código 3.1. En un primer momento, el **identificador_estructura** se emplea para nombrarla y localizarla. Luego, se declaran las variables a utilizar y, en su caso, las funciones correspondientes. Tanto las variables como las funciones que sean declaradas reciben el nombre de *campos de la estructura*. A su vez, las etiquetas **nombre_estructura_1** y **nombre_estructura_2** designan *objetos de la estructura*, entendidos como variables vinculadas con los campos definidos (según algunos autores, también se denominan *variables de la estructura*).

Código 3.1. Sintaxis para declarar una estructura.

```
struct [<identificador_estructura>]
{
   <tipo> <nombre_variable(s)>;
   <tipo> <nombre_variable(s)>;
   <tipo> <definición de función o función prototipo>;
   …
} <nombre_estructura_1>, <nombre_estructura_2>, …;
```

Al comparar la sintaxis del Código 3.1 con el ejemplo del Código 3.2, se puede observar que el nombre de la estructura es **`ejemplo`**. Las variables declaradas al interior corresponden a los tipos **`int`**, **`float`** y **`double`**, mientras que **`obj`** refiere a un objeto de dicha estructura, por medio del cual es posible acceder a sus campos.

Código 3.2. Estructura de `ejemplo`.

```
struct Ejemplo
{
   int i, j;
   float a[5];
   double z[3][4];
} obj;
```

La Figura 3.1 muestra la representación gráfica de la estructura **`ejemplo`**.

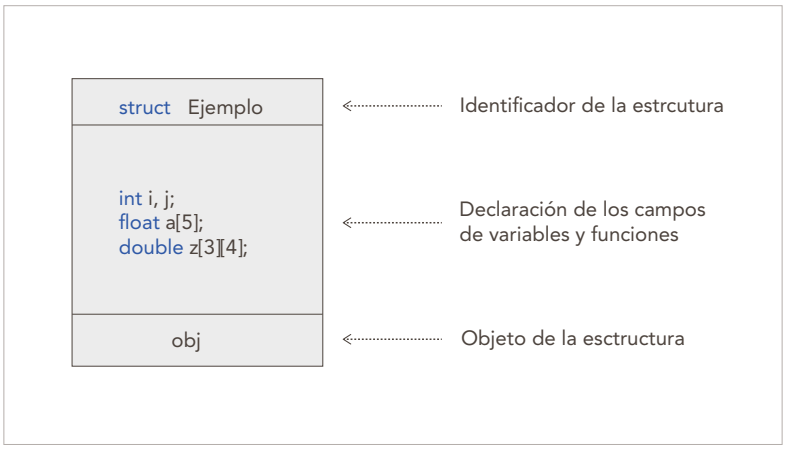


Figura 3.1. Representación gráfica de la estructura `ejemplo`.

Para acceder a un campo dentro de la estructura se escribe el nombre del objeto seguido del carácter de punto (.) y el campo específico, tal como se muestra a continuación:

```
obj.i = 2;
```

Para ilustrar el uso de la estructura **`ejemplo`**, se diseña el Algoritmo 3.1, donde se asignan valores a los campos de la estructura y, posteriormente, se imprimen.

Algoritmo 3.1. Uso de la estructura `ejemplo`.

```
1.      Se define la estructura ejemplo.
2.      Se declara el objeto de la estructura obj.
3.      Se asigna valor al campo i.                    obj.i ← 2
4.      Se establece valor al campo j.                 obj.j ← -5
5.      Se definen valores aleatorios al campo a.      obj.a ← aleatorio
6.      Se inicializan valores aleatorios al campo z.  obj.z ← aleatorio
7.      Se imprimen los valores de todos los campos.
8.      Fin.
```

La Figura 3.2 muestra el diagrama de flujo correspondiente al proceso descrito en el Algoritmo 3.1. En este caso, la función **`rand()`**, común en diversos lenguajes de programación, entrega un valor aleatorio. El diagrama puede representarse de forma esquemática, como se visualiza en la Figura 3.3. Vale decir que no es estrictamente necesario detallar cada paso del procedimiento, sino que puede esbozarse en términos generales. En coherencia con lo planteado, el Código 3.3 presenta la estructura denominada **`ejemplo`**. Al inicio, se definen los campos dentro del bloque, en este caso, variables de tipo entero, flotante y doble; luego, se declara el **`obj`** asociado a la estructura. Por último, dentro de la función **`main()`** se dotan valores a los campos e imprimen los resultados correspondientes.

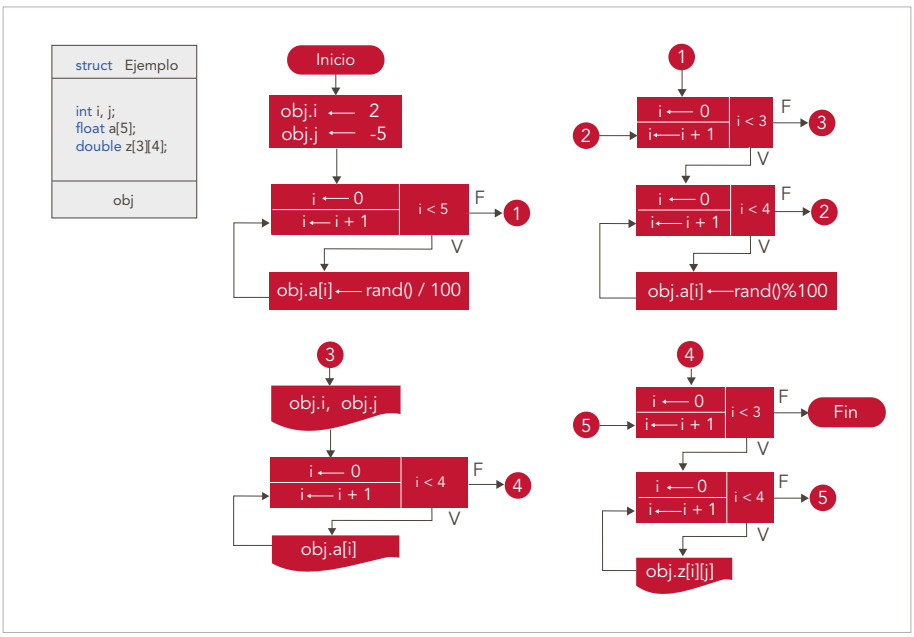


Figura 3.2. Diagrama de flujo para la asignación de valores al objeto de estructura `obj`.

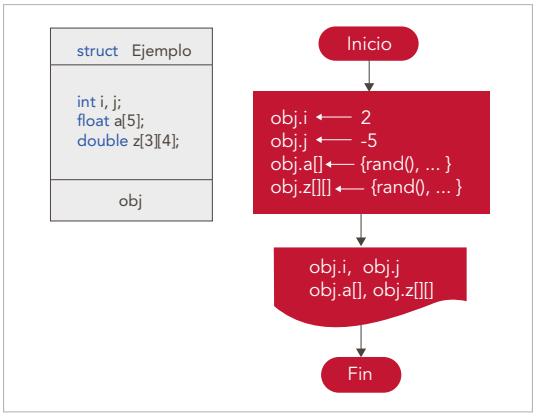


Figura 3.3. Proceso de generación de una estructura.

Código 3.3. Estructura `ejemplo` y acceso a sus `campos`.

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

using namespace std;

struct Ejemplo //Paso 1
{
    int i, j;
    float a[5];
    double z[3][4];
} obj;    //Paso 2

void main()
{
    int i, j;

    //Pasos 3 a 6
    //Se asignan valores a los campos de la estructura
    obj.i = 2;      //Valor al campo i
    obj.j = -5;     //Valor al campo j

    for (i = 0; i < 5; i++)
         obj.a[i] = (float)rand() / (float)100.0;          //Valores al campo a

    for (i = 0; i < 3; i++)
         for (j = 0; j < 4; j++)
              obj.z[i][j] = rand() % 100;       //Valores al campo z

    //Paso 7
    //Se imprimen los valores de los campos de la estructura
    cout << "obj.i = " << obj.i << endl;       //Campo i
    cout << "obj.j = " << obj.j << endl;       //Campo j
    cout << "a[] = ";
```

```
    for (i = 0; i < 5; i++)
         cout << obj.a[i] << "\t";    //Campo a
    cout << endl;
    cout << "z[][] = " << endl;

    for (int i = 0; i < 3; i++)
    {
         for (int j = 0; j < 4; j++)
              cout << obj.z[i][j] << "\t";     //Campo z
         cout << endl;
    }

    //Se termina el proceso con las instrucciones
    _getch();
}
```

La salida del Código 3.3 corresponde a:

```
obj.i = 2
obj.j = -5
a[] = 0.41    184.67  63.34   265     191.69
z[][] =
24      78      58      62
64      5       45      81
27      61      91      95
```

La asociación de la variable a la estructura puede establecerse de forma local o dentro de la función **`main()`**, así como en cualquier otra función. Con base en la estructura **`ejemplo`**, el diagrama de flujo se representa en la Figura 3.4, cuya implementación corresponde a su vez, al Código 3.4.

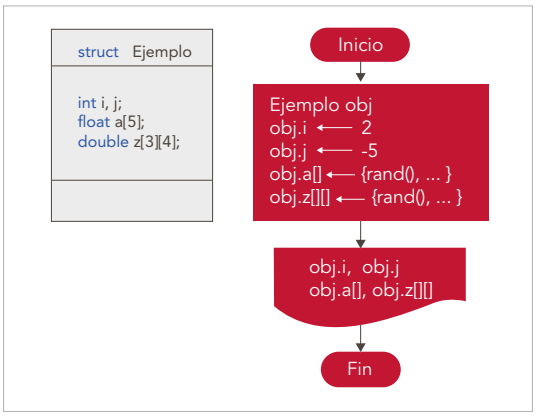


Figura 3.4. Declaración del objeto de la estructura `ejemplo` de manera local.

Código 3.4. Asignación de variable a una estructura de manera local.

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

using namespace std;

struct Ejemplo
{
    int i, j;
    float a[5];
    double z[3][4];
};

void main()
{
    int i, j;      //Declaración local de variables
    Ejemplo obj;   //Asignación de variable a la estructura Ejemplo

    //Se asignan valores a los campos de la estructura
    obj.i = 2;     //Valor al campo i
    obj.j = -5;    //Valor al campo j
    cout << "a[] = ";
    for (i = 0; i < 5; i++)
         obj.a[i] = (float)rand() / (float)100.0;          //Valores al campo a
    cout << "z[][] = " << endl;
    for (i = 0; i < 3; i++)
         for (j = 0; j < 4; j++)
              obj.z[i][j] = rand() % 100;      //Valores al campo z

    //Se imprimen los valores de los campos de la estructura
    cout << "obj.i = " << obj.i << endl;       //Campo i
    cout << "obj.j = " << obj.j << endl;       //Campo j
    for (i = 0; i < 5; i++)
         cout << obj.a[i] << "\t";    //Campo a
    cout << endl;
    for (int i = 0; i < 3; i++)
    {
         for (int j = 0; j < 4; j++)
              cout << obj.z[i][j] << "\t";     //Campo z
         cout << endl;
    }
    //Se termina el proceso con las instrucciones
    _getch();
}
```

Es factible asociar una o más variables a la estructura, como muestra la Figura 3.5, donde se generan de manera local dos objetos de **`ejemplo`**, denominados como **`obj1`** y **`obj2`**. Cada uno es independiente, ya que posee su propia memoria; en otras palabras, los datos de **`obj1`** no afectan a los campos de **`obj2`**. La Figura 3.6 imprime las direcciones de almacenamiento correspondientes al **`obj1`** y **`obj2`**, demostrando que ambos tienen ubicaciones separadas. El Código 3.5 muestra el programa referente a la Figura 3.6.

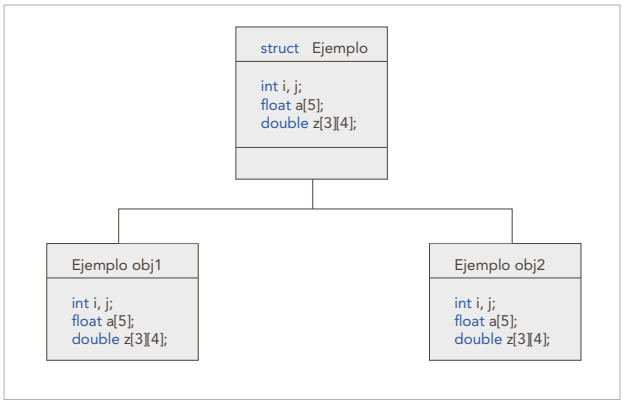


Figura 3.5. Asignación de objetos locales a la estructura `ejemplo`.

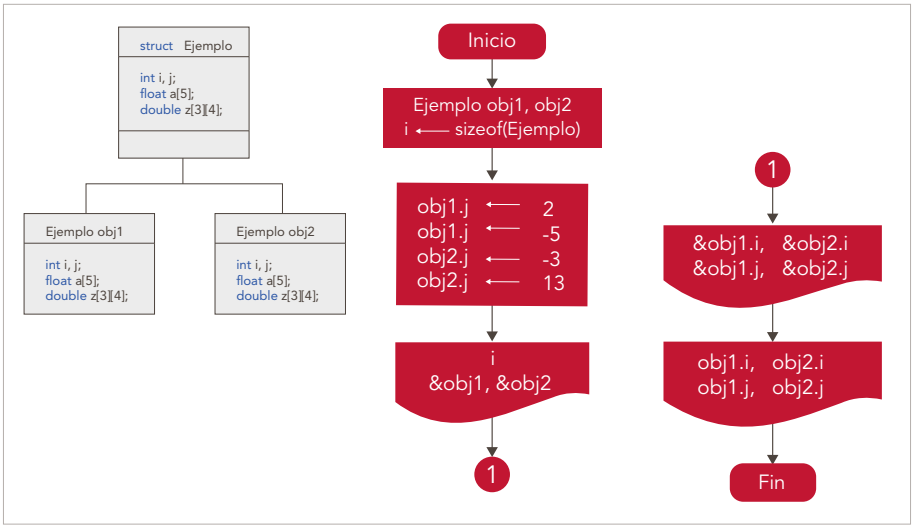


Figura 3.6. Direcciones de memoria asignada a objetos de la misma estructura `ejemplo`.

Código 3.5. Asociación de dos variables a una estructura.

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

using namespace std;

struct Ejemplo
```

```
{
        int i, j;
        float a[5];
        double z[3][4];
};

void main()
{
   int i, j;      //Declaración local de variables
   //Asignación de variables a la estructura Ejemplo
   Ejemplo obj1, obj2;

   //Número de bytes de cada estructura
   i = sizeof(Ejemplo);
   cout << "Tamaño en bytes de Ejemplo = " << i << endl;
   //Se asignan valores a los campos de las estructuras
   obj1.i = 2; obj1.j = -5;
   obj2.i = -3; obj2.j = 13;
   cout << "Direcciones de memoria\n";
   cout << "obj1: " << &obj1 << "\t\tobj2: " << &obj2 << endl;
   cout << "obj1.i: " << &obj1.i << "\tobj2.i: " << &obj2.i << endl;
   cout << "obj1.j: " << &obj1.j << "\tobj2.j: " << &obj2.j << endl;
   cout << "Los datos almacenados son:\n";
   cout << "obj1.i = " << obj1.i << "\tobj1.j = " << obj1.j << endl;
   cout << "obj2.i = " << obj2.i << "\tobj2.j = " << obj2.j << endl;
   _getch();
}
```

La salida del Código 3.5 es:

```
Tamaño en bytes de Ejemplo = 128
Direcciones de memoria
obj1: 00EFFB40          obj2: 00EFFAB8
obj1.i: 00EFFB40        obj2.i: 00EFFAB8
obj1.j: 00EFFB44        obj2.j: 00EFFABC
Los datos almacenados son:
obj1.i = 2      obj1.j = -5
obj2.i = -3     obj2.j = 13
```

El resultado indica que cada objeto de **`ejemplo`** ocupa 128 *bytes* en memoria. Por lo tanto, **`obj1`** requiere 128 *bytes* y **`obj2`** utiliza la misma cantidad. Esto demuestra que los campos de cada estructura son mutuamente excluyentes en términos de almacenamiento, ya que cada variable tiene asignada una dirección de memoria distinta.

3.1.1. INICIALIZAR VALORES DE LOS CAMPOS

Los valores de los campos de una estructura pueden inicializarse al momento de declarar la variable asociada. El Código 3.6 muestra tal proceso:

Código 3.6. Inicialización de variables campo.

```
#include <iostream>
#include <conio.h>
using namespace std;

struct Ejemplo
{
    int i, j;
    float a[5];
    double z[3][4];
};

void main()
{
    int i, j;
    //Asignación de variable a la estructura Ejemplo
    Ejemplo obj1 = { 4, 10, 0.1, 0.2, 0.3, 0.4, 0.5, 1, 2, 3, 4,
         5, 6, 7, 8, 9, 10, 11, 12 };
    Ejemplo obj2 = { 2, -1, 1.1, 1.2, 1.3, 1.4, 1.5, 1.1, 1.2, 1.3,
         1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12 };

    //Se imprimen los valores de los campos de obj1
    cout << "obj1.i = " << obj1.i << endl;    //Campo i
    cout << "obj1.j = " << obj1.j << endl;    //Campo j
    cout << "obj1.a[] = ";
    for (i = 0; i < 5; i++)
         cout << obj1.a[i] << "\t";   //Campo a
    cout << endl;
    cout << "obj1.z[][] =" << endl;
    for (int i = 0; i < 3; i++)
    {
         for (int j = 0; j < 4; j++)
              cout << obj1.z[i][j] << "\t";    //Campo z
         cout << endl;
    }
    cout << "\n\n";
    //Se imprimen los valores de los campos de obj2
    cout << "obj2.i = " << obj2.i << endl;    //Campo i
    cout << "obj2.j = " << obj2.j << endl;    //Campo j
    cout << "obj2.a[] = ";
    for (i = 0; i < 5; i++)
         cout << obj2.a[i] << "\t";   //Campo a
    cout << endl;
    cout << "obj2.z[][] =" << endl;
    for (i = 0; i < 3; i++)
    {
         for (j = 0; j < 4; j++)
              cout << obj2.z[i][j] << "\t";    //Campo z
         cout << endl;
    }
    //Se termina el proceso con las instrucciones
    _getch();
}
```

3.1.2. FUNCIONES EN UNA ESTRUCTURA

Un aspecto característico de las estructuras es que habilitan el despliegue de funciones que ejecutan procesos internos. Por ejemplo, las funciones `asignar()` e `imprimir()`: la primera confiere valores a los campos; la segunda, por su parte, los muestra. La Figura 3.7 presenta la estructura y la definición de sus funciones.

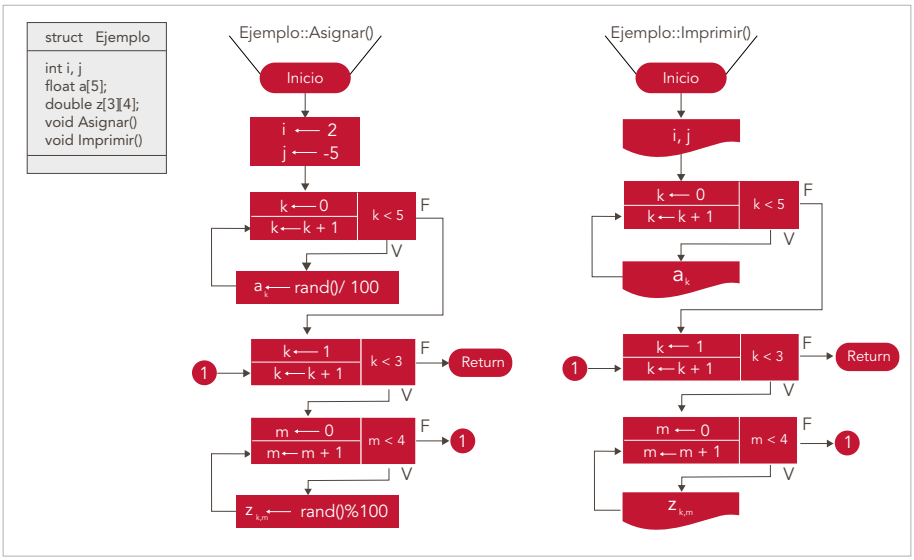


Figura 3.7. Diagrama de flujo simplificado para la estructura `ejemplo`.

La representación de los pasos en un diagrama de flujo, al ser mostrada en reiteradas ocasiones, puede tornarse tediosa y compleja; por ende, se tiene la opción de elaborarla de forma simplificada. La Figura 3.8 muestra el diagrama resumido.

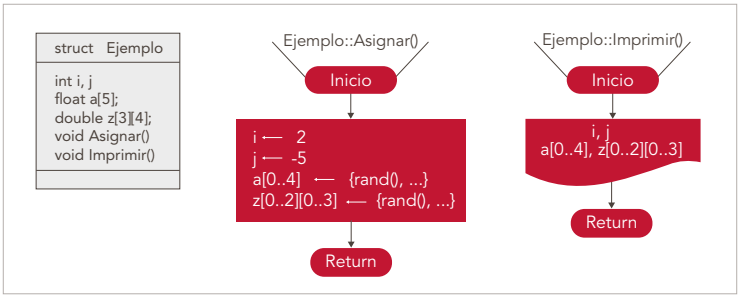


Figura 3.8. Diagrama de flujo resumido de la estructura `ejemplo`.

En el Código 3.7 se declaran dos variables de la estructura **ejemplo**. Para cada variable, es posible establecer una función que asigne valores a los campos y otra que los imprima. De este modo, la tarea se ejecuta con solo invocar la función, sin necesidad de repetir el código para cada objeto.

Código 3.7. Uso de funciones en una estructura para asignar e imprimir valores.

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

using namespace std;

struct Ejemplo
{
    int i, j;
    float a[5];
    double z[3][4];

    void Asignar()
    {
         int k, m;

         i = 0;    //Valor al campo i
         j = 0;    //Valor al campo j
         for (k = 0; k < 5; k++)
              a[k] = (float)rand() / (float)100.0;        //Valores al campo a
         for (k = 0; k < 3; k++)
              for (m = 0; m < 4; m++)
                   z[k][m] = rand() % 100;               //Valores al campo z
    }
    void Imprimir()
    {
         int k, m;
         cout << "i = " << i << endl;          //Campo i
         cout << "j = " << j << endl;          //Campo j
         cout << "a[] = ";
         for (k = 0; k < 5; k++)
              cout << a[k] << "\t";            //Campo a
         cout << endl;
         cout << "z[][] = " << endl;
         for (k = 0; k < 3; k++)
         {
              for (m = 0; m < 4; m++)
                   cout << z[k][m] << "\t";     //Campo z
              cout << endl;
         }
    }
};
void main()
{
    int i, j;
    //Asignación de variable a la estructura Ejemplo
    Ejemplo obj1, obj2;
```

```
    //Se imprimen los valores de los campos de obj1
    obj1.Asignar();
    obj2.Asignar();
    cout << "obj1" << endl;
    obj1.Imprimir();
    cout << "\n\nobj2\n";
    obj2.Imprimir();
    _getch();
}
```

La salida del código es:

```
obj1
i = 0
j = 0
a[] = 0.41      184.67  63.34   265     191.69
z[][] =
24      78      58      62
64      5       45      81
27      61      91      95

obj2
i = 0
j = 0
a[] = 119.42    48.27   54.36   323.91  146.04
z[][] =
2       53      92      82
21      16      18      95
47      26      71      38
```

El manejo de la estructura se lleva a cabo en la función `main()`, donde se asignan los objetos `obj1` y `obj2`. La Figura 3.9 expone el diagrama correspondiente.

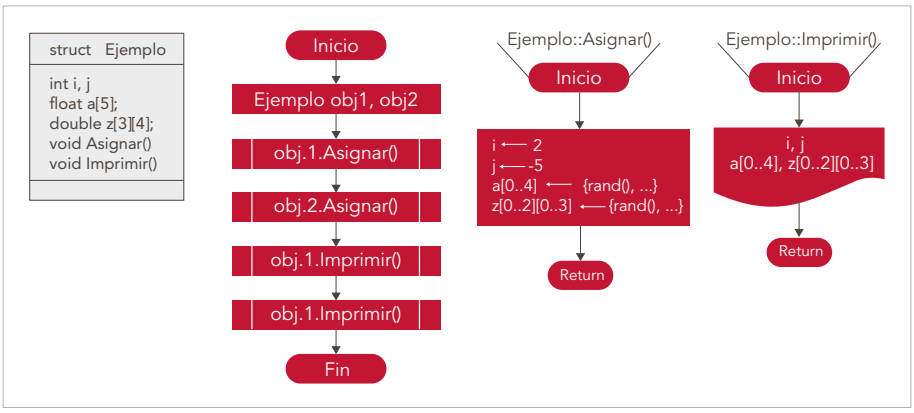


Figura 3.9. Creación y manejo de los objetos de la estructura `ejemplo`.

3.1.3. USO DE FUNCIONES PROTOTIPO EN UNA ESTRUCTURA

Asimismo, las funciones pueden definirse por fuera de la estructura; no obstante, sus prototipos correspondientes deben declararse en el código. Esta acción se ejemplifica en el Código 3.8, cuya salida es similar al resultado del 3.7. La ventaja derivada es la escritura modular del código, lo que facilita el mantenimiento del software.

Código 3.8. Uso de funciones prototipo en una estructura.

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

using namespace std;

struct Ejemplo
{
    int i, j;
    float a[5];
    double z[3][4];

    void Asignar();
    void Imprimir();
};

void Ejemplo::Asignar()
{
    int k, m;

    i = 0;         //Valor al campo i
    j = 0;         //Valor al campo j
    for (k = 0; k < 5; k++)
         a[k] = (float)rand() / (float)100.0;    //Valores al campo a
    for (k = 0; k < 3; k++)
         for (m = 0; m < 4; m++)
              z[k][m] = rand() % 100;          //Valores al campo z
}

void Ejemplo::Imprimir()
{
    int k, m;
    cout << "i = " << i << endl;     //Campo i
    cout << "j = " << j << endl;     //Campo j
    cout << "a[] = ";
    for (k = 0; k < 5; k++)
         cout << a[k] << "\t";        //Campo a
    cout << endl;
    cout << "z[][] = " << endl;
    for (k = 0; k < 3; k++)
    {
         for (m = 0; m < 4; m++)
              cout << z[k][m] << "\t";         //Campo z
         cout << endl;
    }
}
```

```
void main()
{
    int i, j;
    //Asignación de variable a la estructura Ejemplo
    Ejemplo obj1, obj2;

    //Se imprimen los valores de los campos de obj1
    obj1.Asignar();
    obj2.Asignar();
    cout << "obj1" << endl;
    obj1.Imprimir();
    cout << "\n\nobj2\n";
    obj2.Imprimir();
    _getch();
}
```

3.1.4. SOBRECARGA DE FUNCIONES EN UNA ESTRUCTURA

La sobrecarga en C++ refiere a la creación de múltiples funciones que comparten un mismo nombre, pero difieren en sus parámetros. Tal característica permite al programador usar una sola designación para tareas similares, adaptándose a distintos tipos o cantidades de argumentos, lo que facilita la comprensión y mantenimiento del código. Como ejemplo, en el Código 3.8 los campos `i` y `j` se inicializan en cero mediante **`asignar()`**. Por el contrario, el diagrama de flujo de la Figura 3.10 muestra la declaración de dos funciones **`asignar()`**: la primera prescinde de argumentos, por lo cual los campos se inicializan en cero; la segunda acepta dos argumentos para proporcionar valores específicos a los campos. También existe la función **`ciclo()`**, usada para establecer campos que son arreglos. El Código 3.9 corresponde al programa representado en el diagrama de la Figura 3.10.

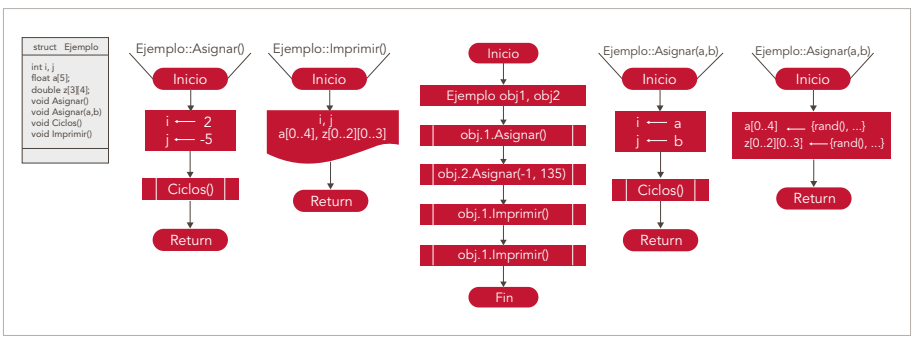


Figura 3.10. Sobrecarga de funciones en la estructura `ejemplo`.

Código 3.9. Sobrecarga de funciones en estructuras.

```
#include <iostream>
#include <conio.h>
using namespace std;

struct Ejemplo
{
    int i, j;
    float a[5];
    double z[3][4];
    void Asignar();
    void Asignar(int a, int b);
    void Ciclos();
    void Imprimir();
};

void Ejemplo::Ciclos()
{
    int k, m;
    for (k = 0; k < 5; k++)
         a[k] = (float)rand() / (float)100.0;    //Valores al campo a
    for (k = 0; k < 3; k++)
         for (m = 0; m < 4; m++)
              z[k][m] = rand() % 100;          //Valores al campo z
}

void Ejemplo::Asignar()
{
    i = 0;         //Valor al campo i
    j = 0;         //Valor al campo j
    Ciclos();
}

void Ejemplo::Asignar(int a, int b)
{
    i = a;         //Valor al campo i
    j = b;         //Valor al campo j
    Ciclos();
}

void Ejemplo::Imprimir()
{
    int k, m;
    cout << "i = " << i << endl;     //Campo i
    cout << "j = " << j << endl;     //Campo j
    cout << "a[] = ";
    for (k = 0; k < 5; k++)
         cout << a[k] << "\t";        //Campo a
    cout << endl;
    cout << "z[][] = " << endl;
    for (k = 0; k < 3; k++)
    {
         for (m = 0; m < 4; m++)
              cout << z[k][m] << "\t";         //Campo z
         cout << endl;
    }
}
```

```
void main()
{
    int i, j;
    //Asignación de objetos a la estructura Ejemplo
    Ejemplo obj1, obj2;
    //Se imprimen los valores de los campos de obj1
    obj1.Asignar();
    obj2.Asignar(-1, 135);
    cout << "obj1" << endl;
    obj1.Imprimir();
    cout << "\n\nobj2\n";
    obj2.Imprimir();
    _getch();
}
```

La salida del Código 3.9 es:

```
obj1
i = 0
j = 0
a[] = 0.41      184.67  63.34   265     191.69
z[][] =
24      78      58      62
64      5       45      81
27      61      91      95


obj2
i = -1
j = 135
a[] = 119.42    48.27   54.36   323.91  146.04
z[][] =
2       53      92      82
21      16      18      95
47      26      71      38
```

3.1.5. FUNCIÓN CONSTRUCTOR

Una vez asimilado el uso de funciones dentro de una estructura, puede abordarse la función *constructor*. Dicho método se aplica tanto para inicializar los campos de la estructura como para poner en marcha algún proceso específico. Se diferencia por carecer de tipo y tener el mismo nombre que la estructura. Su ejecución ocurre una sola vez, al momento de generar el objeto de la estructura; después, no será posible llamarlo nuevamente. La sintaxis correspondiente se muestra en el Código 3.10.

Código 3.10. Sintaxis del constructor en una estructura.

```
struct <identificador_estructura>
{
    <tipo> <nombre_variable(s)>;
    <tipo> <nombre_variable(s)>;
    …
    identificador_estructura();
} <nombre_variable_estructura>, <nombre_variable_estructura>, …;
```

El constructor se activa al crear un objeto para la estructura. En la Figura 3.11 se explaya el diagrama de flujo y su implementación se muestra en el Código 3.11.

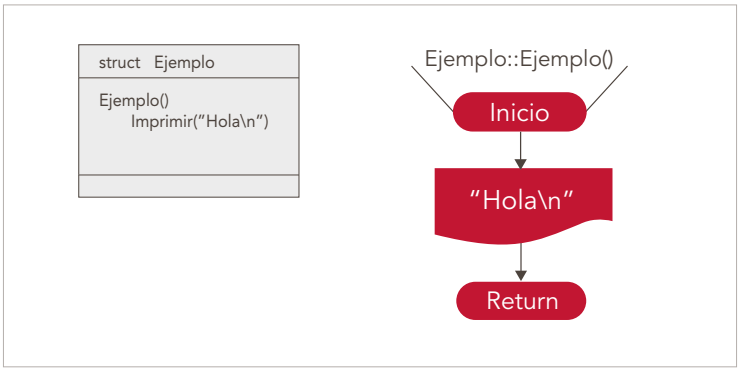


Figura 3.11. Constructor de la estructura `ejemplo`.

Código 3.11. Constructor de la estructura `ejemplo`.

```
struct Ejemplo
{
    Ejemplo()     //Este es el constructor
    {
         printf("Hola\n");
    }
};
```

La Figura 3.12 presenta el diagrama de flujo que ilustra la interacción entre la estructura **`ejemplo`** y la función principal **`main()`**.

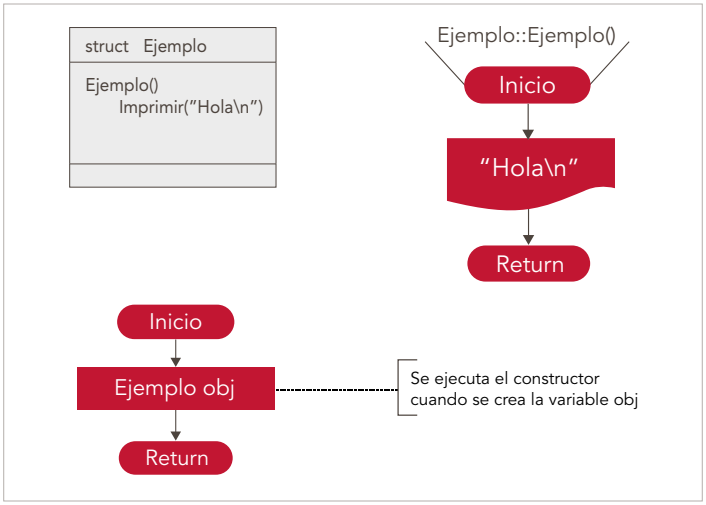


Figura 3.12. Proceso del constructor de una estructura.

El programa correspondiente a la Figura 3.12 se presenta en el Código 3.12. En tal circunstancia, el constructor se ejecuta al asociar la estructura a un objeto, lo que provoca la impresión del mensaje "Hola" en pantalla. Es importante señalar que la función **`ejemplo()`** no se invoca directamente desde **`main()`**. Al crear la variable **`obj`** dentro de **`main()`**, el constructor **`ejemplo()`** se ejecuta de manera automática, cuya única tarea programada consiste en mostrar el mensaje "Hola".

Código 3.12. Función constructor de una estructura.

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

struct Ejemplo
{
    Ejemplo()      //Este es el constructor
    {
         printf("Hola\n");
    }
};

void main()
{
    Ejemplo obj;
    _getch();
}
```

3.1.6. FUNCIÓN DESTRUCTOR

Las estructuras, asimismo, disponen de una función referida como *destructor*, que se ejecuta únicamente cuando la variable asociada a la estructura es eliminada. Tal método se aplica para tareas como la liberación de memoria y el cierre de archivos que han sido previamente abiertos, entre otras operaciones de limpieza. Una función destructor se distingue por ser homónima de la estructura, precedida por el carácter ~. La Figura 3.13 ilustra su representación gráfica.

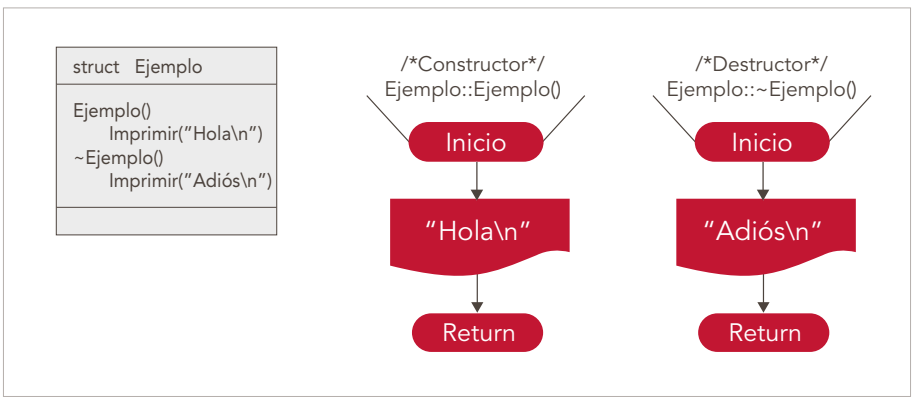


Figura 3.13. Constructor y destructor de la estructura `ejemplo`.

El código que expone el funcionamiento de la Figura 3.13 se encuentra en el Código 3.13.

Código 3.13. Constructor y destructor de la estructura `ejemplo`.

```
struct Ejemplo
{
        Ejemplo()       //Este es el constructor
        {
                printf("Hola\n");
        }

        ~Ejemplo()      //Este es el destructor
        {
                printf("Adiós\n");
        }

};
```

Al finalizar de ejecutarse una función, y eliminarse sus variables locales, se libera la memoria ocupada por cada una. Así, el destructor se invoca en automático en cuanto la variable asociada a una estructura es removida. La Figura 3.14 plasma el funcionamiento del constructor y del destructor, mientras que el Código 3.14 exhibe el programa **`ejemplo`** modificado. A propósito, se convoca la subrutina **`Prueba()`** para crear la variable **`obj`** asociada a **`ejemplo`**. Al generarse **`obj`** de manera local, se ejecuta el constructor antes de proceder a la siguiente instrucción. Al concluir la subrutina, la variable es depuesta, activándose de este modo la función destructor.

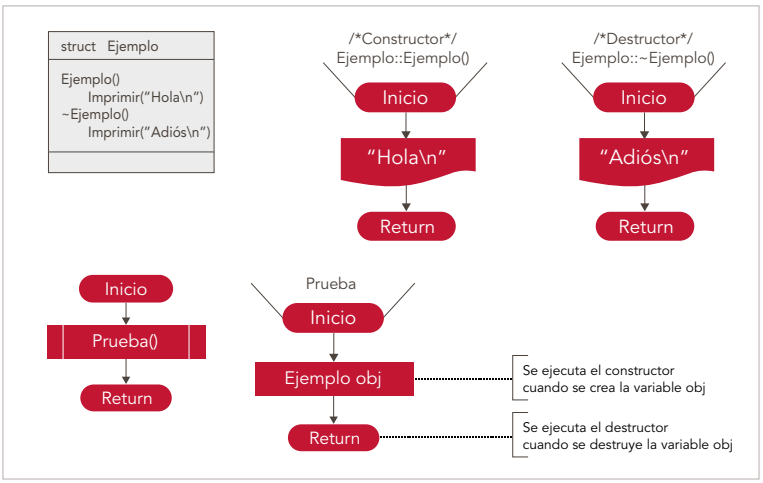


Figura 3.14. Proceso de llamado del constructor y del destructor.

Código 3.14. Constructor y destructor de una estructura.

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

struct Ejemplo
{
    //Este es el constructor
    Ejemplo()
    {
         printf("Hola\n");
    }

    //Este es el destructor
    ~Ejemplo()

    {
         printf("Adiós\n");
    }
```

```
};

void Prueba()
{
    Ejemplo obj;
}

void main()
{
    Prueba();
    _getch();
}
```

3.1.7. SOBRECARGA DEL CONSTRUCTOR

El constructor puede sobrecargarse cuando existen dos o más variables homónimas que difieren en la cantidad o tipo de argumentos. El diagrama de flujo de la Figura 3.15 expone la estructura **`ejemplo`**, donde el constructor inicializa los campos y el destructor imprime sus valores. El primero se encuentra sobrecargado en dos versiones: una no recibe argumentos, a diferencia de la otra, que acepta dos para inicializar los campos.

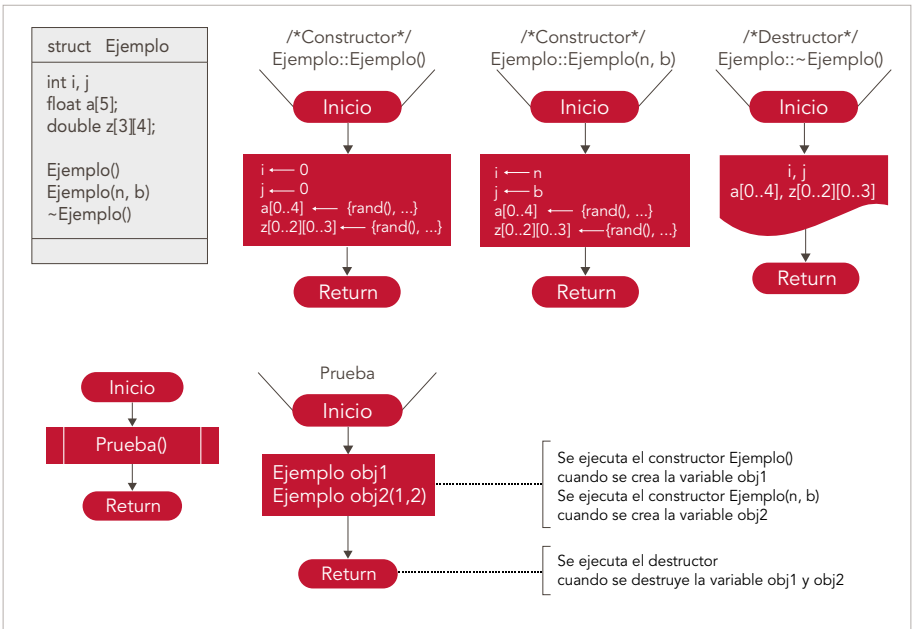


Figura 3.15. Sobrecarga del constructor de la estructura `Ejemplo`.

El programa representado en la Figura 3.15 se muestra en el Código 3.15.

Código 3.15. Sobrecarga del constructor.

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

using namespace std;

struct Ejemplo
{
    int i, j;
    float a[5];
    double z[3][4];

    Ejemplo();
    Ejemplo(int n, int b);
    ~Ejemplo();
};

Ejemplo::Ejemplo()
{
    int k, m;

    i = 0;          //Valor al campo i
    j = 0;          //Valor al campo j
    for (k = 0; k < 5; k++)
    a[k] = (float)rand() / (float)100.0;      //Valores al campo a
    for (k = 0; k < 3; k++)
    for (m = 0; m < 4; m++)
         z[k][m] = rand() % 100;              //Valores al campo z
}

Ejemplo::Ejemplo(int n, int b)
{
    int k, m;

    i = n;          //Valor al campo i
    j = b;          //Valor al campo j
    for (k = 0; k < 5; k++)
    a[k] = (float)rand() / (float)100.0;      //Valores al campo a
    for (k = 0; k < 3; k++)
         for (m = 0; m < 4; m++)
              z[k][m] = rand() % 100;         //Valores al campo z
}

Ejemplo::~Ejemplo()
{
    int k, m;
    cout << "i = " << i << endl;     //Campo i
    cout << "j = " << j << endl;     //Campo j
    cout << "a[] = ";
    for (k = 0; k < 5; k++)
         cout << a[k] << "\t";       //Campo a
```

```
    cout << endl;
    cout << "z[][] = " << endl;
    for (k = 0; k < 3; k++)
    {
         for (m = 0; m < 4; m++)
              cout << z[k][m] << "\t";         //Campo z
         cout << endl;
    }
}

void Prueba()
{
    Ejemplo obj1, obj2(1, 2);
}

void main()
{
    int i, j;
    Prueba();
    _getch();
}
```

La salida del programa es:

```
i = 1
j = 2
a[] = 119.42    48.27   54.36   323.91  146.04
z[][] =
2       53      92      82
21      16      18      95
47      26      71      38

i = 0
j = 0
a[] = 0.41      184.67  63.34   265     191.69
z[][] =
24      78      58      62
64      5       45      81
27      61      91      95
```

3.1.8. ASIGNACIÓN DE ESTRUCTURAS

Es posible asignar un objeto a otro, siempre que ambos pertenezcan a la misma estructura y tipo. La Figura 3.16 ilustra el proceso mediante una estructura **`ejemplo`**, con dos campos enteros **`i`** y **`j`**. En el Código 3.16 se denota cómo la variable **`obj2`** recibe el contenido de otro objeto: sus campos adoptan los valores correspondientes de **`obj1`**.

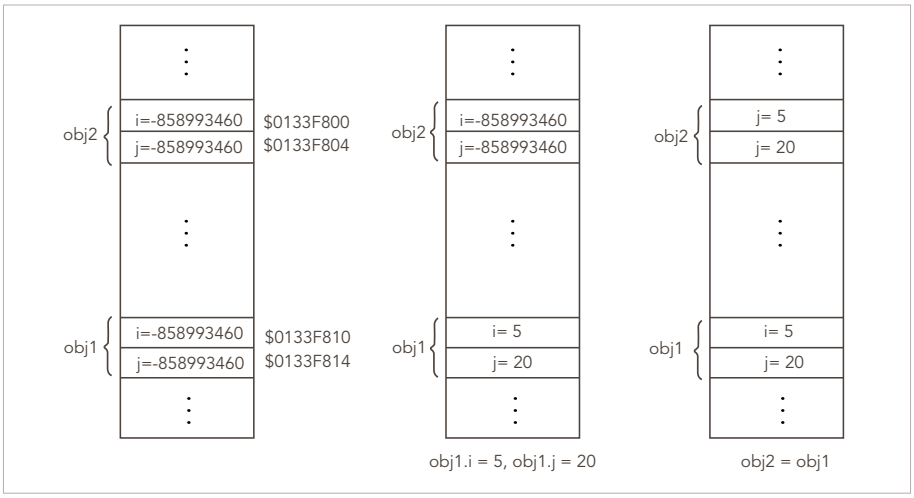


Figura 3.16. Asignación entre objetos de una estructura.

Código 3.16. Asignación de estructuras.

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

using namespace std;

struct Ejemplo
{
   int i, j;
};

void main()
{
   Ejemplo obj1, obj2;

   obj1.i = 5;
   obj1.j = 20;
   obj2 = obj1;
   printf("obj2.i = %d, obj2.j = %d\n", obj2.i, obj2.j);
   _getch();
}
```

La salida es:

```
obj2.i = 5, obj2.j = 20
```

3.1.9. ARREGLO DE ESTRUCTURAS

Es viable declarar un arreglo de estructuras donde cada elemento posea sus propios campos independientes, tal como se muestra en el Código 3.17.

Código 3.17. Arreglo de estructuras.

```
#include <iostream>
#include <conio.h>
#include <stdio.h>
using namespace std;

struct Ejemplo
{
    int i;
    float k;
    double j;
};

void main()
{
    Ejemplo obj[3];
    for (int i = 0; i < 3; i++)
    {
         obj[i].i = (int)rand() % 10;
         obj[i].k = sqrt(rand() / 1000);
         obj[i].j = rand() / 55.33;
    }
    //Imprimiendo direcciones de memoria
    for (int i = 0; i < 3; i++)
    {
         cout << "&obj[" << i << "] = " << &obj[i] << "\t";
         cout << "&obj[" << i << "].i = " << &obj[i].i << "\t";
         cout << "&obj[" << i << "].k = " << &obj[i].k << "\t";
         cout << "&obj[" << i << "].j = " << &obj[i].j << endl;
    }
    //Imprimiendo los valores
    for (int i = 0; i < 3; i++)
    {
         cout << "obj[" << i << "].i = " << obj[i].i << "\t";
         cout << "obj[" << i << "].k = " << obj[i].k << "\t";
         cout << "obj[" << i << "].j = " << obj[i].j << endl;
    }
    _getch();
}
```

El resultado visible en pantalla es similar a:

```
&obj[0] = 0055FCB4    &obj[0].i = 0055FCB4  &obj[0].k = 0055FCB8   &obj[0].j = 0055FCBC
&obj[1] = 0055FCC4    &obj[1].i = 0055FCC4  &obj[1].k = 0055FCC8   &obj[1].j = 0055FCCC
&obj[2] = 0055FCD4    &obj[2].i = 0055FCD4  &obj[2].k = 0055FCD8   &obj[2].j = 0055FCDC
```

```
obj[0].i = 1    obj[0].k = 4.24264      obj[0].j = 114.477
obj[1].i = 0    obj[1].k = 4.3589       obj[1].j = 284.186
obj[2].i = 8    obj[2].k = 5.38516      obj[2].j = 487.294
```

Al analizar el mapa de memoria, se advierte que `obj` ocupa un espacio de memoria que va de `$0055FCB4` a `$0055FCE3`, como se ilustra en la Figura 3.17.

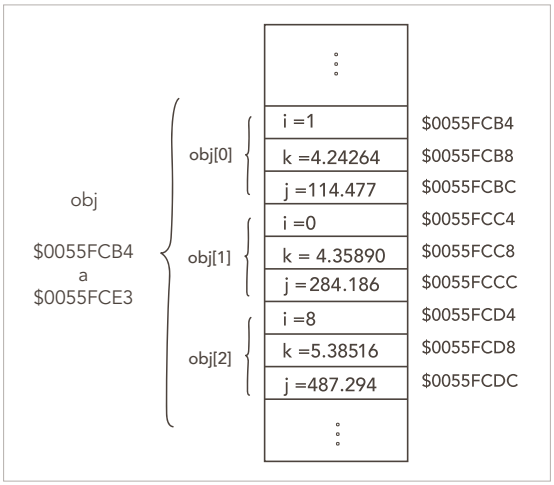


Figura 3.17. Memoria asignada para el ejemplo del Código 3.17.

3.1.10. ESTRUCTURAS ANIDADAS

En términos generales, las estructuras anidadas —como sugiere su nombre— son aquellas que contienen otras estructuras. En cierto modo, pueden compararse con una *matrioshka*, muñeca rusa que alberga en su interior varias muñecas de menor tamaño. Por consiguiente, tales construcciones admiten combinar múltiples tipos de información y representarla de forma flexible al almacenar datos dentro de datos, conformando así estratos complejos. En exceso, su empleo puede resultar contraproducente y derivar en una mala *praxis*. En el peor de los casos, el anidamiento impide reutilizar estructuras en diferentes contextos, complica la lectura, la comprensión y la depuración del código; asimismo, aumenta el acoplamiento entre componentes y la dependencia de unos con otros. Aunado a esto, tal tipo de estructura puede generar sobrecarga en la memoria y perjudicar el rendimiento del programa. Por estas razones se desaconseja el anidamiento de estructuras, salvo que exista una justificación plausible para implementarlo.

3.1.11. APUNTADORES A ESTRUCTURAS

Los campos de una estructura pueden accederse para leer o escribir datos mediante apuntadores. En el Código 3.18 se designa un apuntador `p` para acceder a los campos de `obj` y, así, otorgar valores a `i` y `j`. Resulta posible accesar de dos maneras distintas, identificadas en el código como `forma 1` y `forma 2`.

Código 3.18. Apuntadores a estructuras.

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

struct Ejemplo
{
    int i, j;
};

void main()
{
    Ejemplo obj, * p;
    p = &obj;
    (*p).i = 15;   //Forma 1
    p->j = 23;     //Forma 2
    printf("obj.i = %d, obj.j = %d\n", obj.i, obj.j);
    _getch();
}
```

3.1.12. EJERCICIO: OPERACIONES CON NÚMEROS COMPLEJOS MEDIANTE ESTRUCTURAS

Comprensión del problema

Los números complejos constituyen un conjunto numérico fundamental en campos como las matemáticas, la física y la ingeniería; se representan como la suma de una parte real y un múltiplo real de la unidad imaginaria *i*. Por ejemplo:

$$3 + 4i, -5 + 8i, 1.5 - 4.75i, \ldots$$

En términos generales, el número complejo A puede representarse como:

$$A = B + Ci$$

Donde:

B y C números reales,
i unidad imaginaria ($\sqrt{-1}$)

Las operaciones que pueden efectuarse con números complejos corresponden a las básicas de la aritmética: suma, resta, multiplicación y división.

Si se consideran los complejos $Y = a + bi$ y $X = c + di$, las operaciones básicas se definen de la siguiente forma:

Suma:

$$Y + X = (a + bi) + (c + di)$$
$$Y + X = (a + c) + (b + d)i$$

Resta:

$$Y - X = (a + bi) - (c + di)$$
$$Y - X = (a - c) + (b - d)i$$

Multiplicación:

$$Y \times X = (a + bi) \times (c + di) = ac + adi + cbi - bd$$
$$Y \times X = (ac - bd) + (ad + cb)i$$

División:

$$\frac{Y}{X} = \frac{a+bi}{c+di} = \frac{a+bi}{c+di} \times \frac{c-di}{c-di} = \frac{(ac+bd)+(bc-ad)i}{c^2+d^2}$$
$$\frac{Y}{X} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i$$

Algoritmo

El procedimiento para resolver el problema se plantea en el Algoritmo 3.1, el cual se presenta de manera simplificada.

Algoritmo 3.1. Operaciones básicas con números complejos.

```
1. Definir la estructura.
2. Pedir los valores de las partes real e imaginaria para Y y X.
3. Preguntar el tipo de operación a realizar.
4. Ejecutar la operación elegida.
5. Imprimir el resultado.
```

Diagrama de flujo

La Figura 3.18 ilustra el diagrama correspondiente a la definición de la estructura; a su vez, el diagrama de flujo del programa principal se presenta en la Figura 3.19.

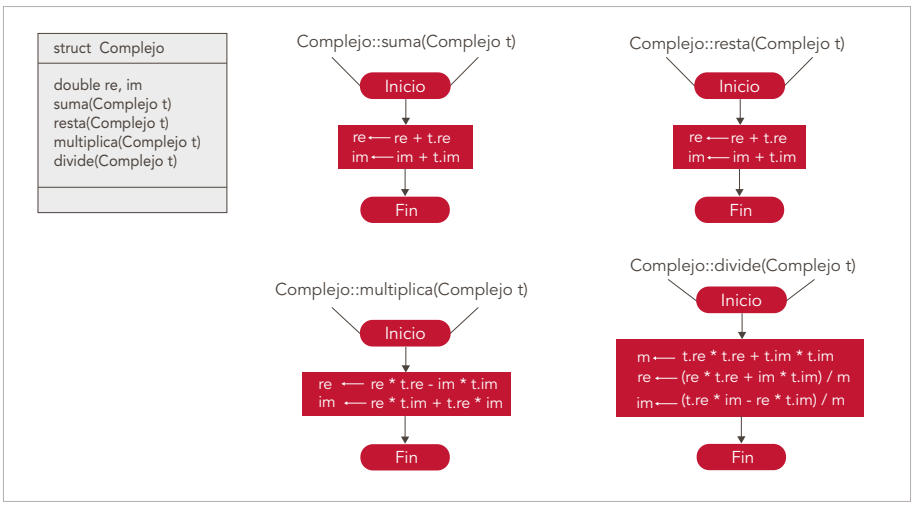


Figura 3.18. Definición de la estructura `complejo`.

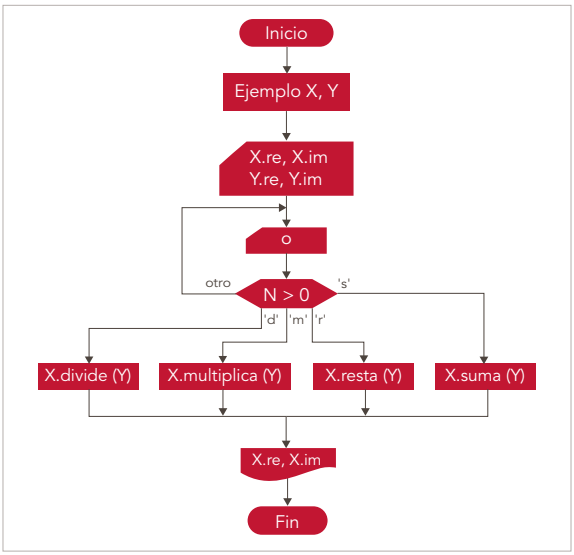


Figura 3.19. Diagrama de flujo del programa principal.

Código en C++

Una vez fraguado el método para resolver el problema, se procede a desarrollar el programa mostrado en el Código 3.19.

Código 3.19. Operaciones con números complejos.

```
#include <conio.h>

//Manejo de números complejos
struct Complejo {
    double re, im;
    Complejo();
    void suma(Complejo t);
    void resta(Complejo t);
    void multiplica(Complejo t);
    void divide(Complejo t);
};

Complejo::Complejo()
{
    re = im = 0.0;
}

void Complejo::suma(Complejo t)
{
    re += t.re;
    im += t.im;
}

void Complejo::resta(Complejo t)
{
    re -= t.re;
    im -= t.im;
}

void Complejo::multiplica(Complejo t)
{
    re = re * t.re - im * t.im;
    im = re * t.im + t.re * im;
}

void Complejo::divide(Complejo t)
{
    double m;
    m = t.re * t.re + t.im * t.im;
    re = (re * t.re + im * t.im) / m;
    im = (t.re * im - re * t.im) / m;
}

void main()
{
    char o;
    Complejo X, Y;
```

```
    X.re = 1; X.im = 2;
    Y.re = -2; Y.im = 1;
    printf("Qué operación desea realizar?\n");
    printf("<s> Suma\n<r> Resta\n<m> Multiplicación\n");
    printf("<d> División");
    do
    {
         o = _getch();
    } while (o != '' && o != 'r' && o != 'm' && o != 'd');
    switch (o)
    {
    case 's': X.suma(Y); break;
    case 'r': X.resta(Y); break;
    case 'm': X.multiplica(Y); break;
    case 'd': X.divide(Y);
    }
    printf("Y = %f + %fi\n", X.re, X.im);
    _getch();
}
```

3.1.13. SOBRECARGA DE OPERADORES PARA ESTRUCTURAS

Una característica prominente de C++ es la sobrecarga de operadores, recurso que permite a una misma operación, como la adición, adaptar su comportamiento dependiendo del tipo de datos: no es lo mismo sumar enteros que valores flotantes o dobles. El compilador identifica el tipo de operación deseada y la ejecuta. La Figura 3.20 ejemplifica este proceso en un diagrama de flujo. Sin embargo, solo algunos lenguajes permiten personalizar la sobrecarga de operadores. Asimismo, conviene recordar que los operadores pueden clasificarse como binarios o unarios, según cuántos argumentos requiere la operación.

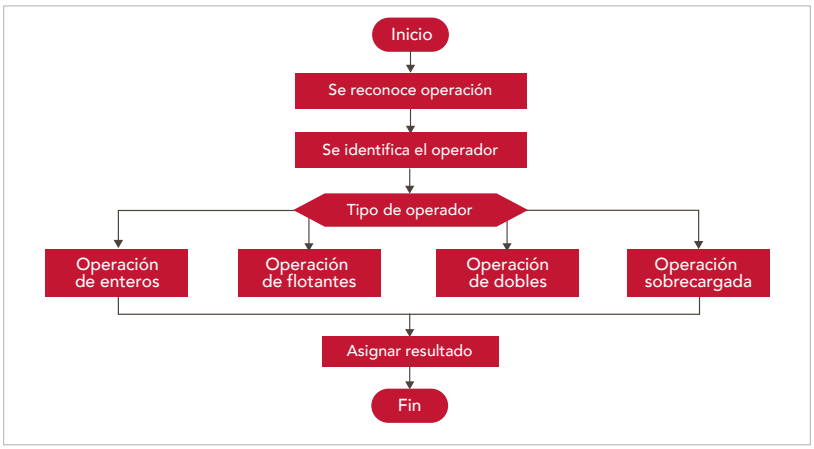


Figura 3.20. Diagrama de flujo de identificación de operador y tipo de operación.

3.1.14. SOBRECARGA DE OPERADORES BINARIOS PARA ESTRUCTURAS

Para sobrecargar un operador binario, se implementa la sintaxis del Código 3.20. El elemento **<tipo>** corresponde al identificador de la estructura de variable sobre la cual se ejecutará la operación. Por otro lado, **<operador>** representa el símbolo del tipo de operación que se desea sobrecargar. Ahora bien, para realizarla, los **<argumentos>** constituyen los valores necesarios, mientras que **<sentencias>** equivale a los comandos.

Código 3.20. Sintaxis para sobrecargar un operador binario.

```
<tipo> operator <operador> (<argumentos>)
{
   <sentencias>;
}
```

Los operadores sobrecargables dentro de una estructura son los siguientes:

- **+** Suma dos valores.
- **–** Resta el valor del operando derecho al del operando izquierdo.
- ***** Multiplica dos valores.
- **/** Divide el valor de la izquierda por el valor de la derecha.
- **%** Calcula el residuo de la división del valor de la izquierda por el valor de la derecha.
- **^** Operador XOR (bitwise exclusive OR) para operaciones de bits.
- **&** Operador AND (bitwise AND) para operaciones de bits.
- **|** Operador OR (bitwise OR) para operaciones de bits.
- **!** Operador de negación lógica (NOT). Invierte el valor lógico de la expresión; devuelve **true** si la condición es **false**, y viceversa.
- **==** Comprueba si dos valores son iguales.
- **+=** Incrementa el valor de la izquierda sumándole el valor de la derecha y almacena el resultado en la variable de la izquierda.
- **–=** Decrementa el valor de la izquierda restando el valor de la derecha y almacena el resultado en la variable de la izquierda.

`*=`	Multiplica el valor de la izquierda por el valor de la derecha y almacena el resultado en la variable izquierda.
`/=`	Divide el valor de la izquierda por el valor de la derecha y almacena el resultado en la variable izquierda.
`!=`	Comprueba si dos valores no son iguales.
`<`	Corrobora si el valor de la izquierda es menor que el valor de la derecha.
`<=`	Revisa si el valor de la izquierda es menor o igual al valor de la derecha.
`>`	Constata si el valor de la izquierda es mayor que el valor de la derecha.
`>=`	Examina si el valor de la izquierda es mayor o igual al valor de la derecha.

El Código 3.21 implementa la sobrecarga del operador `+` para la estructura `complejo`. Una representación gráfica de este proceso se exhibe en la Figura 3.21.

Código 3.21. Sobrecarga del operador + para la estructura `complejo`.

```
struct complejo
{
    float r, i;
    void Print() {cout << r << " + " << i << "i" << endl;}
};

// Sobrecarga del operador + para complejo
complejo operator +(complejo a, complejo b) {
    complejo res = { a.r + b.r, a.i + b.i };
    return res;
}
```

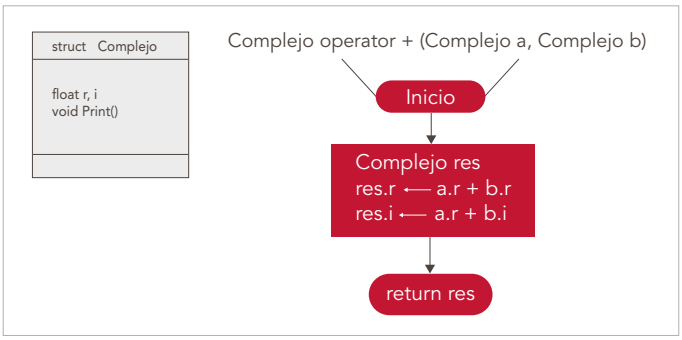


Figura 3.21. Representación gráfica de la sobrecarga de operadores para estructuras.

La Figura 3.22 desglosa la sobrecarga de cuatro operadores, así como su aplicación dentro de la función principal `main()`. El programa implementado se presenta en el Código 3.22.

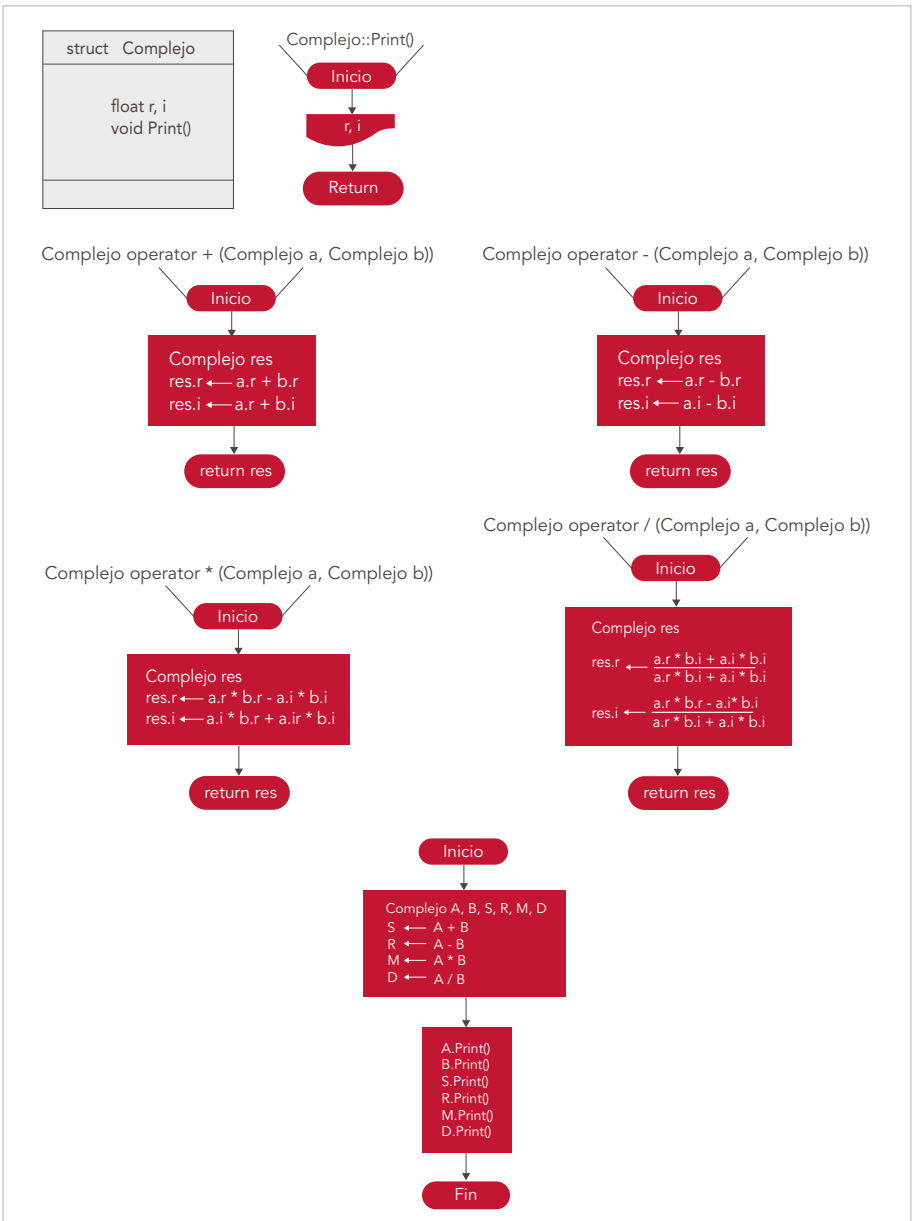


Figura 3.22. Sobrecarga de cuatro operadores para la estructura `complejo`.

Código 3.22. Aplicación de la sobrecarga de operadores para la estructura `complejo`.

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

struct complejo
{
    float r, i;
    void Print() {cout << r << " + " << i << "i" << endl;}
};

// Sobrecarga del operador + para complejo
complejo operator +(complejo a, complejo b) {
    complejo res = { a.r + b.r, a.i + b.i };
    return res;
}

// Sobrecarga del operador - para complejo
complejo operator -(complejo a, complejo b) {
    complejo res = { a.r - b.r, a.i - b.i };
    return res;
}

// Sobrecarga del operador * para complejo
complejo operator *(complejo a, complejo b) {
    complejo res = { a.r * b.r - a.i * b.i, a.i * b.r + a.r * b.i };
    return res;
}

// Sobrecarga del operador / para complejo
complejo operator /(complejo a, complejo b) {
    complejo res;
    res.r = (a.r * b.r + a.i * b.i) / (b.r * b.r + b.i * b.i);
    res.i = (a.i * b.r - a.r * b.i) / (b.r * b.r + b.i * b.i);
    return res;
}

int main()
{
    complejo A = { 2, 3 };
    complejo B = { -1, 4 };
    complejo S, R, M, D;

    S = A + B;
    R = A - B;
    M = A * B;
    D = A / B;
    cout << "A = "; A.Print();
    cout << "B = "; B.Print();
    cout << "S = "; S.Print();
    cout << "R = "; R.Print();
    cout << "M = "; M.Print();
    cout << "D = "; D.Print();
    return 1;
}
```

El resultado de la ejecución del Código 3.22 es el siguiente:

```
A = 2 + 3i
B = -1 + 4i
S = 1 + 7i
R = 3 + -1i
M = -14 + 5i
D = 0.588235 + -0.647059i
```

Los operadores de comparación, como **==**, también pueden sobrecargarse. El Código 3.23 expone la implementación correspondiente.

Código 3.23. Sobrecarga del operador == para estructuras.

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

struct complejo
{
   float r, i;
   void Print() {cout << r << " + " << i << "i" << endl;}
};

// Sobrecarga del operador == para complejo
int operator ==(complejo a, Complejo b)
{
   if ((a.r == b.r) && (a.i == b.i)) return 1;
   else return 0;
}

int main()
{
   complejo A = { 2, 3 };
   complejo B = { -1, 4 };
   complejo C = { 2,3 };

   if (A == B)
        cout << "A = B" << endl;
   else
        cout << "A != B" << endl;

   if (A == C)
        cout << "A = B" << endl;
   else
        cout << "A != C" << endl;
   return 1;
}
```

El resultado equivale a:

```
A != B
A = B
```

Asimismo, es posible efectuar operaciones con argumentos diferentes. El Código 3.24 ejemplifica la sobrecarga del operador *, donde uno de los argumentos posee un valor de 0.5.

Código 3.24. Sobrecarga del operador * con un argumento tipo `float`.

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

struct complejo
{
    float r, i;
    void Print() {cout << r << " + " << i << "i" << endl;}
};

// Sobrecarga del operador * para complejo
Complejo operator *(float a, complejo b)
{
    return { b.r * a, b.i * a };
}

int main()
{
    complejo A = { -1, 4 };
    complejo B;

    B = 0.5 * A;
    cout << "A = "; A.Print();
    cout << "3 * A = "; B.Print();

    _getch();
    return 1;
}
```

El resultado es:

```
A = -1 + 4i
3 * A = -0.5 + 2i
```

Sin embargo, si la operación se cambia por `B = A * 0.5`, el compilador generará un error, puesto que desconoce el orden de los argumentos. Por último, es relevante mencionar que la sobrecarga de operadores no presenta ninguna restricción en cuanto al tipo de operador ni de argumentos.

3.1.15. SOBRECARGA DE OPERADORES UNARIOS PARA ESTRUCTURAS

Los operadores unarios sobrecargables son `++` y `--`, y ambos pueden fungir como prefijo o sufijo, según sea su posición respecto a la variable. Al momento en que se colocan como sufijo, el operador se ubica a la derecha de la variable; en contraste, como prefijo, se encuentra a la izquierda. El Código 3.25 muestra la sobrecarga del operador `++` en posición de prefijo, la cual incrementa el valor del objeto `A` de tipo `complejo`.

Código 3.25. Uso del operador ++ como prefijo, no devuelve valor alguno.

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

struct complejo
{
   float r, i;
   void Print() {cout << r << " + " << i << "i" << endl;}
};

// Sobrecarga del operador prefijo ++ para complejo
void operator ++(complejo& a)
{
   a.r += 1; a.i += 1;
}

void main()
{
   complejo A = { 2, 3 };

   A.Print();
   ++A;//Prefijo
   A.Print();
}
```

El resultado es:

```
2 + 3i
3 + 4i
```

En consonancia con lo anterior, es viable sobrecargar el operador de manera que el resultado pueda asignarse a otro objeto del mismo tipo. Cabe señalar que la instrucción `C = ++A` implica, en primera instancia, incrementar el valor del objeto `A` y, después, realizar la asignación. Tal comportamiento se exhibe en el Código 3.26.

Código 3.26. La sobrecarga del operador ++ como prefijo devuelve un dato `complejo`.

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

struct complejo
{
    float r, i;
    void Print() {cout << r << " + " << i << "i" << endl;}
};

// Sobrecarga del operador prefijo ++ para complejo
complejo operator ++(complejo& a)
{
    complejo res;
    res = { ++a.r, ++a.i };
    return res;
}

int main()
{
    complejo A = { 2, 3 };
    complejo C;

    cout << "A = "; A.Print();
    C = ++A;//Prefijo
    cout << "A = "; A.Print();
    cout << "C = "; C.Print();
    return 1;
}
```

El resultado es:

```
A = 2 + 3i
A = 3 + 4i
C = 3 + 4i
```

Cuando el operador se implementa como sufijo, primero se realiza la asignación de valores y, posteriormente, se efectúa el incremento. El Código 3.27 muestra la sobrecarga del operador –– en posición de sufijo.

Código 3.27. Sobrecarga del operador –– como sufijo.

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

struct complejo
{
    float r, i;
    void Print() { cout << r << " + " << i << "i" << endl; }
};
```

```
// Sobrecarga del operador prefijo ++ para complejo
complejo operator --(complejo& a, int i)
{
    complejo res;
    res = { a.r--, a.i-- };
    return res;
}

void main()
{
    complejo A = { 2, 3 };
    complejo C;

    cout << "A = "; A.Print();
    C = A--;       //Prefijo
    cout << "A = "; A.Print();
    cout << "C = "; C.Print();
}
```

3.2. CLASES Y MIEMBROS

Entre las características que potencian al lenguaje C++ se encuentran las clases. Por ello, son ampliamente populares en el desarrollo de plataformas y en la creación de otros lenguajes de programación. Aunque las clases guardan similitud con las estructuras, ofrecen ventajas que fortalecen las capacidades del lenguaje.

3.2.1. DECLARACIÓN DE CLASES

La sintaxis correspondiente a la declaración de una clase se presenta en el Código 3.28.

Código 3.28. Sintaxis de la declaración de una clase.

```
class [<identificador_clase>]
{
<especificador_acceso:>
    <tipo> <nombre_variable(s)>;
    <tipo> <nombre_variable(s)>;
    <tipo> <definición de función o función prototipo>;
    …
} <nombre_objeto_1>, <nombre_objeto_2>, …;
```

La sintaxis para declarar una clase es similar a la de una estructura. Entre sus diferencias se encuentra que las variables y funciones dentro de una clase se denominan *miembros*. Además, el proceso de crear un objeto a partir de una clase se conoce como *instanciación*. Todos los miembros deben contar con un especificador de acceso, que se describe en el siguiente apartado, pero la definición del objeto al final de la declaración de la clase es opcional.

3.2.2. ESPECIFICADORES DE ACCESO

C++ invoca especificadores de acceso para regular la libertad con que se ingresa a los miembros de una clase. En este sentido, existen tres tipos principales: **`public`** (público), **`private`** (privado) y **`protected`** (protegido). Cuando un miembro se declara como **`public`**, puede accederse a él desde cualquier sección del código. Por su parte, los miembros en **`protected`** son accesibles por funciones vinculadas a la misma clase o sus derivadas. De forma inversa, los miembros en **`private`** permanecen inaccesibles desde todas las partes del programa que sean externas a la misma clase, incluidas las clases derivadas, lo que refuerza el principio de encapsulamiento. El Código 3.29 denota la clase **`Estudiante`**, la cual contiene información sobre un hipotético alumno. A este respecto, se restringió la información haciendo uso de los tres tipos de especificadores.

Código 3.29. Definición de la clase `Estudiante` sin objeto instanciado.

```
class Estudiante
{
protected:
   char Nombre[70];
private:
   char email[50];
   int edad;
public:
   char Expediente[6];
   int Asist;
   int totalAsist;
} z;
```

3.2.3. USO DE LAS CLASES

A modo de ejemplo, se comenzará por definir la clase **`Estudiante`**, disponiendo toda la información como pública (**`public`**). Si se omite el especificador de acceso, el compilador lo

identifica, por defecto, como privado. Al definir la clase, es posible instanciar un objeto al final de la declaración, como expone el Código 3.30, para que pueda reconocerse de manera global. Asimismo, es factible definir la clase sin objeto instanciado, como presenta el Código 3.31, a fin de que el objeto opere de manera global o local según sea necesario.

Código 3.30. Definición de la clase `Estudiante` con objeto instanciado.

```
class Estudiante
{
public:
    char Nombre[70];
    char email[50];
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
} est;
```

Código 3.31. Definición de la clase `Estudiante` sin objeto instanciado.

```
class Estudiante
{
public:
    char Nombre[70];
    char email[50];
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
};
```

La Figura 3.23 manifiesta la representación gráfica de ambos tipos de definición de una clase. Dicha interacción puede realizarse desde la función `main()` o cualquier otra que conforme el programa. Aun así, es menester considerar que, si el objeto es instanciado dentro de una función, su alcance será local, es decir, solo podrá utilizarse en esa función. En cambio, si el objeto se instancia junto con la definición de clase, su disponibilidad tendrá alcance global y podrá emplearse en cualquier parte del código.

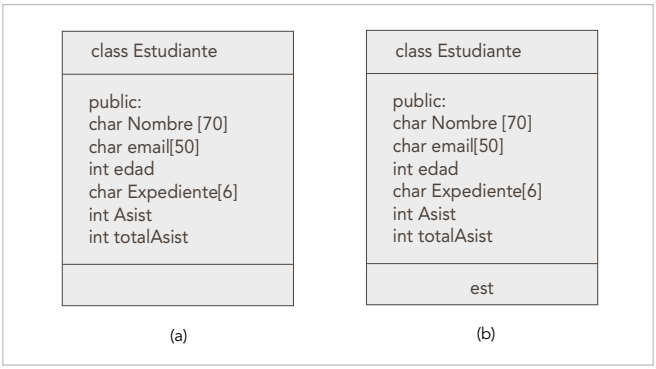


Figura 3.23. Representación gráfica de la clase `Estudiante`.
(a) Sin objeto instanciado. (b) Con objeto instanciado.

Este tipo de gestión de clases se exhibe en el diagrama de flujo de la Figura 3.24, donde el objeto se instancia en la función **`main()`**, por lo que actúa como una variable local. En oposición, en la Figura 3.25, el objeto se instancia al final de la definición de la clase, por lo tanto constituye una variable global, lo que favorece su disponibilidad en cualquier parte del programa. El programa que corresponde a la Figura 3.24 se define en el Código 3.32; en cambio, el de la Figura 3.25 se expone en el Código 3.33. Cabe mencionar que la función **`strcpy_s`** asigna una cadena de caracteres a una variable (también existe el comando **`strcpy`**, pero Visual Studio 2019 lo rechaza por considerarlo inseguro).

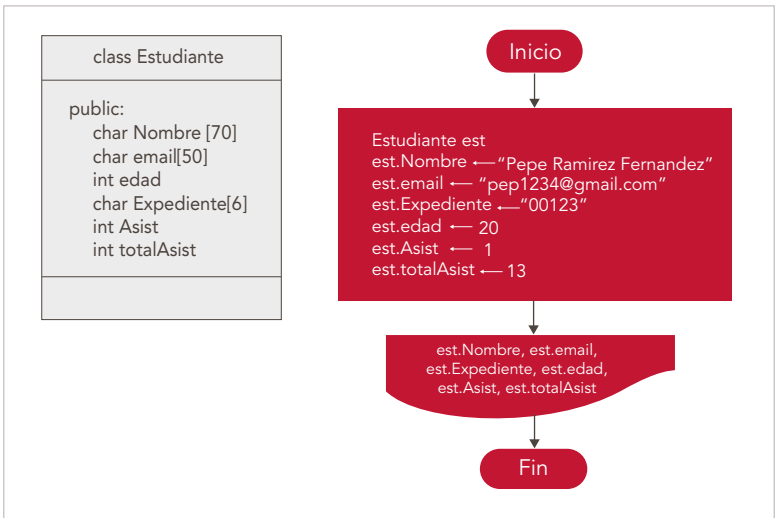


Figura 3.24. Uso de la clase `Estudiante` con el objeto local en `main()`.

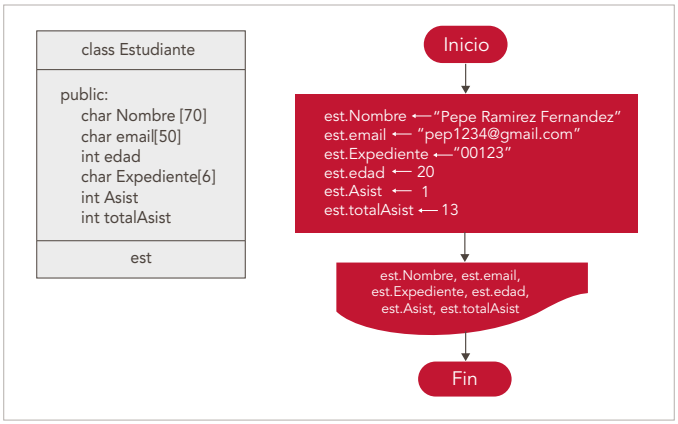


Figura 3.25. Uso de la clase `Estudiante` con el objeto global.

Código 3.32. Uso de la clase `Estudiante` con el objeto como local.

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

class Estudiante
{
public:
    char Nombre[70];
    char email[50];
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
};

void main()
{
    Estudiante est;

    strcpy_s(est.Nombre, "Pepe Ramirez Fernandez");
    strcpy_s(est.email, "pep1234@gmail.com");
    strcpy_s(est.Expediente, "00123");
    est.edad = 20;
    est.Asist = 1;
    est.totalAsist = 13;

    printf("    Nombre: %s\n", est.Nombre);
    printf("     email: %s\n", est.email);
    printf("Expediente: %s\n", est.Expediente);
    printf("      Edad: %d\n", est.edad);
    printf("Asistencia: %d\n", est.Asist);
    printf("Total de asistencias: %d\n", est.totalAsist);
}
```

Código 3.33. Uso de la clase `Estudiante` como global.

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

class Estudiante
{
public:
    char Nombre[70];
    char email[50];
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
} est;

void main()
{
    strcpy_s(est.Nombre, "Pepe Ramirez Fernandez");
    strcpy_s(est.email, "pep1234@gmail.com");
    strcpy_s(est.Expediente, "00123");
    est.edad = 20;
    est.Asist = 1;
    est.totalAsist = 13;

    printf("    Nombre: %s\n", est.Nombre);
    printf("     email: %s\n", est.email);
    printf("Expediente: %s\n", est.Expediente);
    printf("      Edad: %d\n", est.edad);
    printf("Asistencia: %d\n", est.Asist);
    printf("Total de asistencias: %d\n", est.totalAsist);
}
```

La salida generada para los casos de los Códigos 3.32 y 3.33 es la siguiente:

```
Nombre: Pepe Ramirez Fernandez
email: pep1234@gmail.com
Expediente: 00123
Edad: 20
Asistencia: 1
Total de asistencias: 13
```

3.2.4. USO DE FUNCIONES MIEMBRO

Como también ocurre con las estructuras, en las clases es posible declarar funciones miembro que ejecuten diversas tareas. Por ejemplo, la Figura 3.26 presenta el diagrama donde se incorpora la función `Print()`, para imprimir los valores de las variables miembro. El programa acorde se muestra en el Código 3.34.

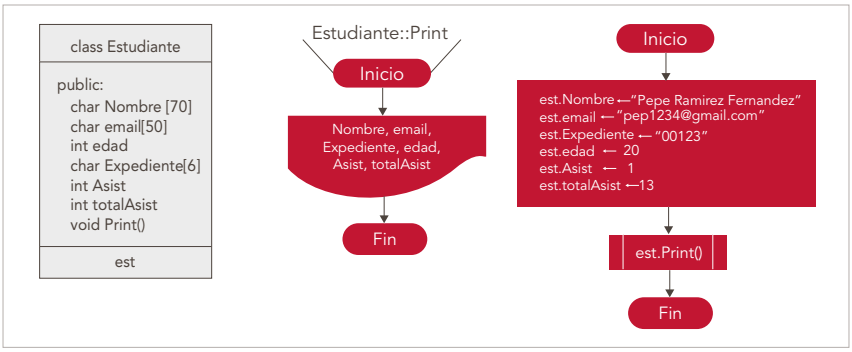


Figura 3.26. Uso de funciones miembro en la clase `Estudiante`.

Código 3.34. Uso de la función miembro `Print()` en la clase `Estudiante`.

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

class Estudiante
{
public:
    char Nombre[70];
    char email[50];
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
    void Print()
    {
         printf("    Nombre: %s\n", Nombre);
         printf("     email: %s\n", email);
         printf("Expediente: %s\n", Expediente);
         printf("      Edad: %d\n", edad);
         printf("Asistencia: %d\n", Asist);
         printf("Total de asistencias: %d\n", totalAsist);
}
} est;

void main()
{
    strcpy_s(est.Nombre, "Pepe Ramirez Fernandez");
    strcpy_s(est.email, "pep1234@gmail.com");
    strcpy_s(est.Expediente, "00123");
    est.edad = 20;
    est.Asist = 1;
    est.totalAsist = 13;
    est.Print();
}
```

Acorde con lo anterior, la función `Print()`también puede definirse fuera de la clase; sin embargo, es menester vincularla a la clase a través del operador de resolución de ámbito denotado por : : . Tal procedimiento se presenta en el Código 3.35.

Código 3.35. Definición de la función miembro usando el operador de ámbito.

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

class Estudiante
{
public:
   char Nombre[70];
   char email[50];
   int edad;
   char Expediente[6];
   int Asist;
   int totalAsist;
   void Print();         //Prototipo de la función miembro
} est;

//Definición de la función miembro de la clase Estudiante
//Se debe usar el operador de resolución de ámbito
void Estudiante::Print()
{
   printf("    Nombre: %s\n", Nombre);
   printf("     email: %s\n", email);
   printf("Expediente: %s\n", Expediente);
   printf("      Edad: %d\n", edad);
   printf("Asistencia: %d\n", Asist);
   printf("Total de asistencias: %d\n", totalAsist);
}

void main()
{
   strcpy_s(est.Nombre, "Pepe Ramirez Fernandez");
   strcpy_s(est.email, "pep1234@gmail.com");
   strcpy_s(est.Expediente, "00123");
   est.edad = 20;
   est.Asist = 1;
   est.totalAsist = 13;
   est.Print();
   _getch();
}
```

3.2.5. USO DEL ESPECIFICADOR DE ACCESO `protected`

Cuando los miembros de una clase se declaran bajo el especificador de acceso **`protected`**, únicamente podrán operarse desde otras funciones miembro de la misma clase y sus derivadas. Con relación a la clase **`Estudiante`**, si tanto la variable **`Nombre`** como **`email`** cambian de **`public`** a **`protected`**, será necesario implementar una función que asigne valores a tales variables miembro. La Figura 3.27 muestra el diagrama de flujo pertinente, mientras que el Código 3.36 presenta el programa.

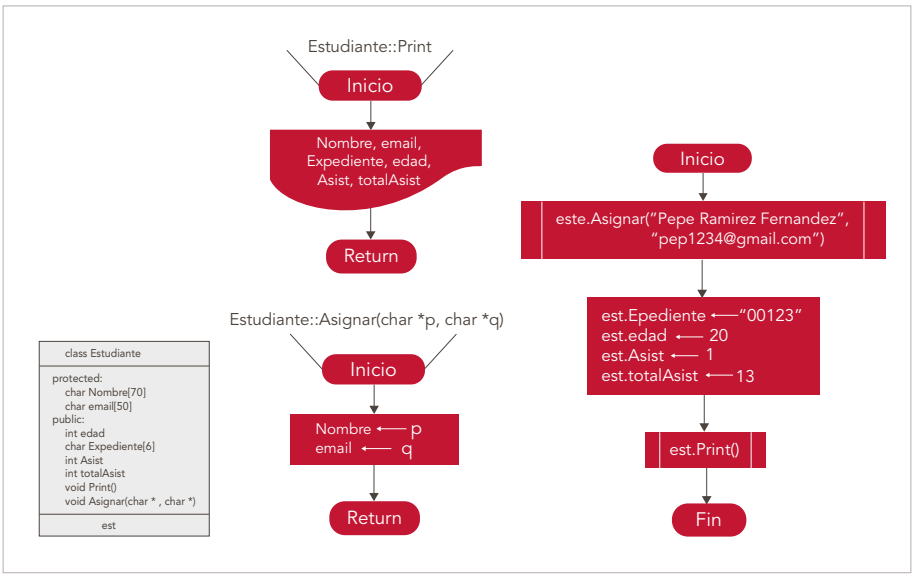


Figura 3.27. Aplicación del especificador de acceso protected en la clase `Estudiante`.

Código 3.36. Asignación de datos a miembro `protected`.

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

class Estudiante
{
protected:
    char Nombre[70];
    char email[50];
```

```
public:
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
    void Print();//Prototipo de la función miembro
    void Asignar(char* p, char* q); //Prototipo de la función para asignar
                            //datos a variables miembro protected
} est;

//Definición de la función miembro de la clase Estudiante
//Se debe usar el operador de resolución de ámbito
void Estudiante::Print()
{
    printf("    Nombre: %s\n", Nombre);
    printf("     email: %s\n", email);
    printf("Expediente: %s\n", Expediente);
    printf("      Edad: %d\n", edad);
    printf("Asistencia: %d\n", Asist);
    printf("Total de asistencias: %d\n", totalAsist);
}

//Definición de la función miembro de la clase Estudiante para
//asignar datos a los miembros protected Nombre e email.
void Estudiante::Asignar(char* p, char* q)
{
    strcpy_s(Nombre, p); //Se asigna dato a una variable miembro protected
    strcpy_s(email, q);  //Se asigna dato a una variable miembro protected
}

void main()
{
    est.Asignar((char*)"Pepe Ramirez Fernandez",
         (char*)"pep1234@gmail.com");
    strcpy_s(est.Expediente, "00123");
    est.edad = 20;
    est.Asist = 1;
    est.totalAsist = 13;
    est.Print();
    _getch();
}
```

3.2.6. CONSTRUCTOR Y DESTRUCTOR DE CLASE

El **constructor** y el **destructor** de una clase se ejecutan una sola vez, y su invocación es automática. Del mismo modo que en las estructuras, el **constructor** de una clase carece de tipo y es homónimo. Su rol es inicializar las variables miembro o disparar algún proceso. El **destructor** también prescinde de tipo y comparte el nombre de la clase, pero se distingue por llevar antepuesto el símbolo tilde (~). Se trata de una función que se aplica expresamente para liberar la memoria ocupada por un objeto: si este es global, se elimina al concluirse la

ejecución del programa. Por el contrario, de ser local, su existencia está limitada a la de su bloque de instrucciones asociado. La Figura 3.28 esquematiza el proceso de ejecución del `constructor` y `destructor`, desde el momento en el cual el objeto es instanciado a la clase hasta su destrucción, conforme a las condiciones descritas anteriormente.

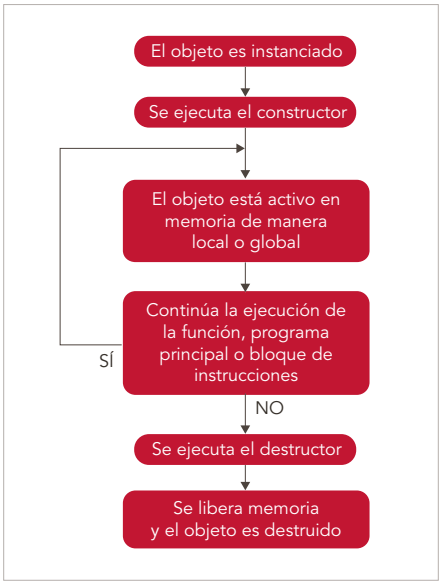


Figura 3.28. Proceso de ejecución del `constructor` y `destructor`.

La Figura 3.29 presenta el diagrama de flujo de la clase `Ejemplo`, destacando los papeles del `constructor` y el `destructor`. El Código 3.37 exhibe la implementación correspondiente del programa.

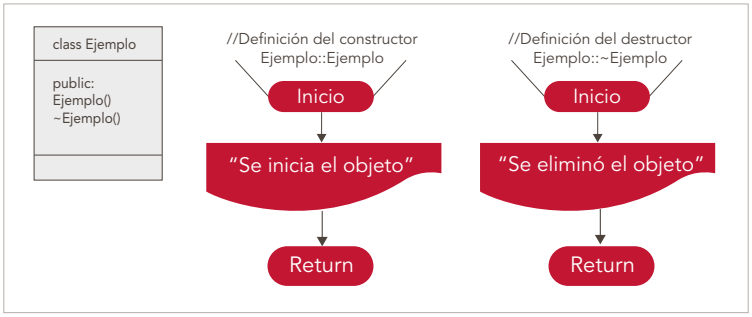


Figura 3.29. Definiciones del `constructor` y el `destructor` de una clase.

Código 3.37. `constructor` y `destructor` para la clase `Ejemplo`.

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

using namespace std;

class Ejemplo
{
public:
    Ejemplo(){cout << "Se inicia el objeto\n";}
    ~Ejemplo(){cout << "Se eliminó el objeto e\n";}
};

void main()
{
    //Se inicia un bloque de instrucciones
    {
         //Se instancia el objeto e a la clase Ejemplo y se ejecuta el constructor
         //El objeto es local solamente en este bloque
         Ejemplo e;
    }     //Se termina el bloque de instrucciones, por lo que se ejecuta el destructor
         //Se elimina el objeto y se libera la memoria.
    _getch();
}
```

Constructor de una clase

Con anterioridad se mencionó que el **`constructor`** se usa para establecer puntos de partida limpios tanto para variables como procesos. Este paso es fundamental, ya que si las variables miembro no se inicializan, adoptarán valores basura, como se ejemplifica a continuación:

```
Nombre: ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠B¶┐⌂T■¤
email: ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠B¶┐
⌂T■¤
Expediente: ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠B¶┐⌂T■¤
Edad: -858993460
Asistencia: -858993460
Total de asistencias: -858993460
```

La Figura 3.30 presenta un diagrama de flujo que ilustra el uso del **`constructor`** para inicializar los miembros de una clase.

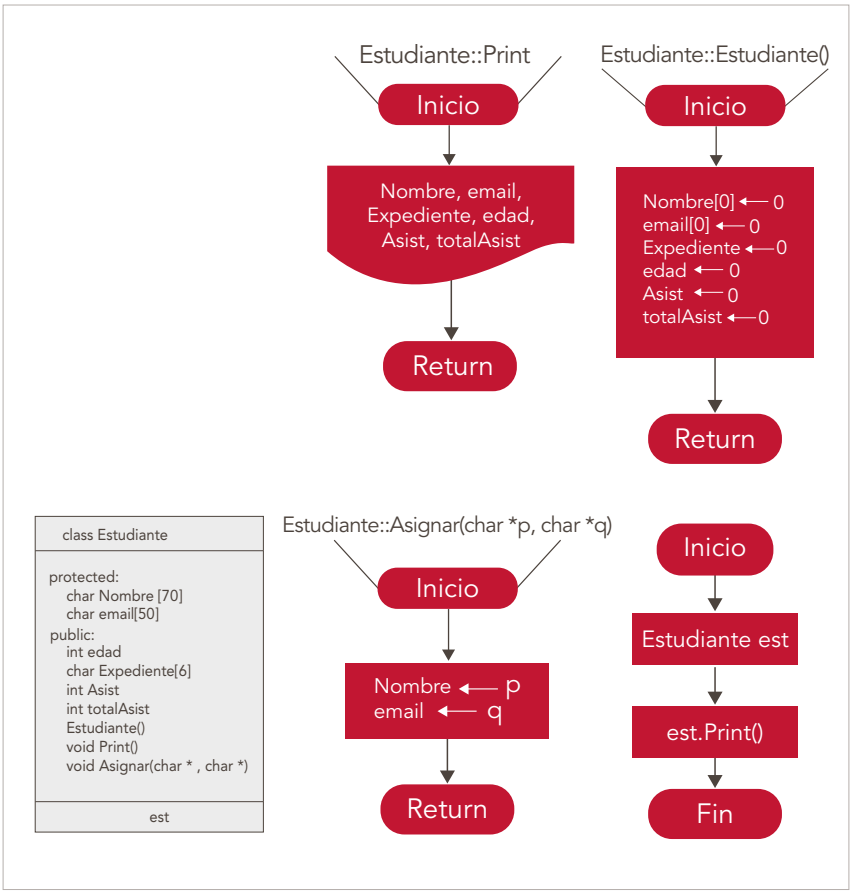


Figura 3.30. Uso del `constructor` en la clase `Estudiante`.

Código 3.38. Empleo del `constructor` en la clase `Estudiante`.

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

class Estudiante
{
protected:
        char Nombre[70];
        char email[50];
public:
        int edad;
```

```
    char Expediente[6];
    int Asist;
    int totalAsist;
    Estudiante();          //Prototipo del constructor
    void Print();          //Prototipo de la función miembro
    void Asignar(char* p, char* q);  //Prototipo de la función para asignar
                           //datos a variables miembro private
};

{
    Nombre[0] = 0;         //Se inicializa con el caracter nulo
    email[0] = 0;          //Se inicializa con el caracter nulo
    edad = 0;
    Expediente[0] = 0;     //Se inicializa con el caracter nulo
    Asist = 0;
    totalAsist = 0;        //Se iniciliza el total de asistencias a 0
}

//Definición de la función miembro de la clase Estudiante
//Se debe usar el operador de resolución de ámbito
void Estudiante::Print()
{
    printf("    Nombre: %s\n", Nombre);
    printf("     email: %s\n", email);
    printf("Expediente: %s\n", Expediente);
    printf("      Edad: %d\n", edad);
    printf("Asistencia: %d\n", Asist);
    printf("Total de asistencias: %d\n", totalAsist);
}

//Definición de la función miembro de la clase Estudiante para
//asignar datos a los miembros protected Nombre e email.
void Estudiante::Asignar(char* p, char* q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void main()
{
    Estudiante est;
    est.Print();
}
```

El **constructor** también puede recibir argumentos, como se muestra en la Figura 3.31 y en el Código 3.39. Al instanciar un objeto a la clase, el **constructor** se ejecuta de forma automática, utilizando el argumento recibido para inicializar la variable **Nombre**.

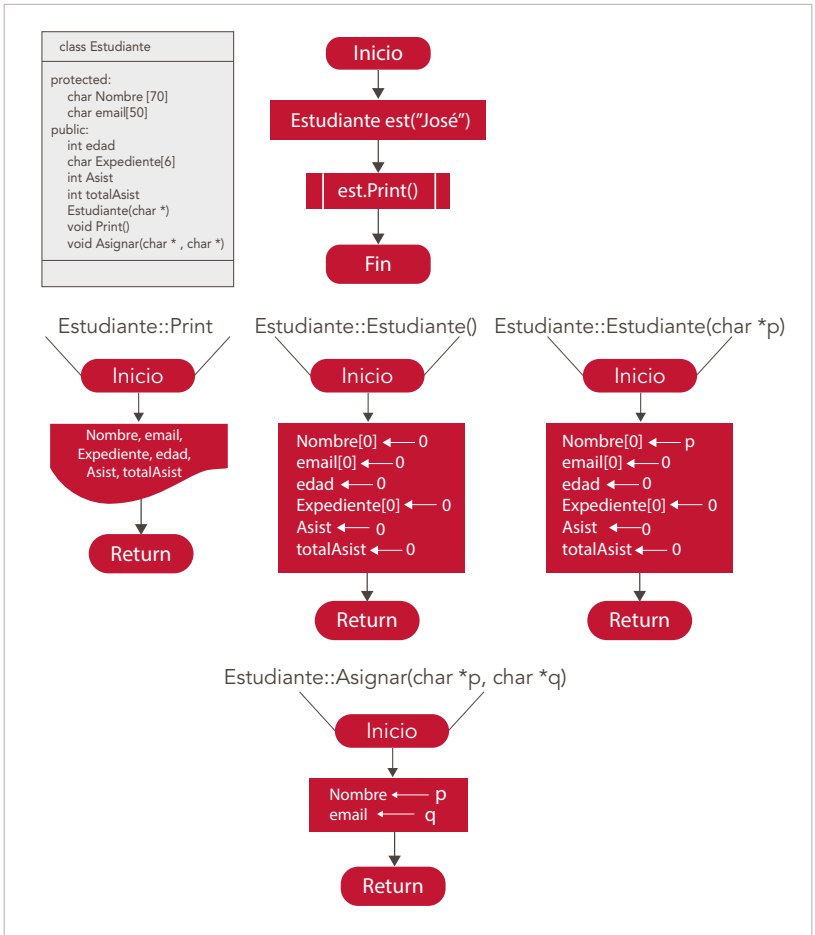


Figura 3.31. Uso del `constructor` con argumento para la clase `Estudiante`.

Código 3.39. Aplicación del `constructor` con argumentos.

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

class Estudiante
{
protected:
    char Nombre[70];
    char email[50];
public:
```

```
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
    Estudiante(char* p);    //Prototipo del constructor
    void Print();           //Prototipo de la función miembro
    void Asignar(char* p, char* q);  //Prototipo de la función para asignar
                            //datos a variables miembro private
};

//Definición del constructor
Estudiante::Estudiante(char* p)
{
    strcpy_s(Nombre, p);    //Se inicializa con el caracter nulo
    email[0] = 0;           //Se inicializa con el caracter nulo
    edad = 0;
    Expediente[0] = 0;      //Se inicializa con el caracter nulo
    Asist = 0;
    totalAsist = 0;         //Se iniciliza el total de asistencias a 0
}

//Definición de la función miembro de la clase Estudiante
//Se debe usar el operador de resolución de ámbito
void Estudiante::Print()
{
    printf("    Nombre: %s\n", Nombre);
    printf("     email: %s\n", email);
    printf("Expediente: %s\n", Expediente);
    printf("      Edad: %d\n", edad);
    printf("Asistencia: %d\n", Asist);
    printf("Total de asistencias: %d\n", totalAsist);
}

//Definición de la función miembro de la clase Estudiante para
//asignar datos a los miembros protected Nombre e email.
void Estudiante::Asignar(char* p, char* q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void main()
{
    Estudiante est((char*)"José");
    est.Print();
}
```

Sobrecarga del constructor

Las funciones constructor de una clase pueden sobrecargarse al igual que otras funciones, como se aprecia en la Figura 3.32.

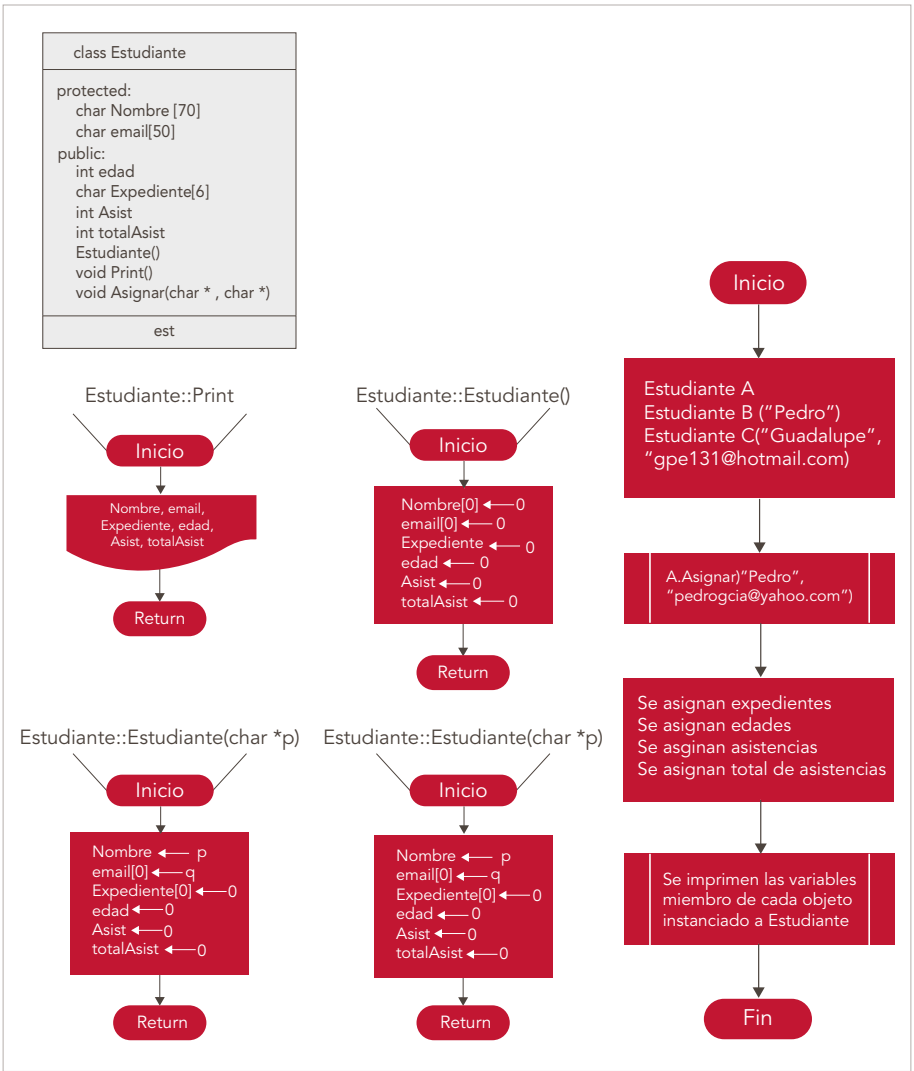


Figura 3.32. Sobrecarga del constructor para la clase `Estudiante`.

La piedra angular de la sobrecarga de funciones son los argumentos. Ahora bien, el diagrama de flujo de la Figura 3.32 ilustra la sobrecarga del constructor en tres ocasiones: primero prescinde de argumentos, en segunda admite solo uno y en tercera incorpora dos. El Código 3.40 presenta la implementación correspondiente del diagrama de flujo.

Código 3.40. Sobrecarga del constructor para `Estudiante`.

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

class Estudiante
{
protected:
    char Nombre[70];
    char email[50];
public:
    int edad;
    char Expediente[6];
    int Asist;
    int totalAsist;
    Estudiante();           //Prototipo del constructor
    Estudiante(char* p);    //Prototipo del constructor
    Estudiante(char* p, char* q);    //Prototipo del constructor
    void Print();           //Prototipo de la función miembro
    void Asignar(char* p, char* q);  //Prototipo de la función para asignar
                            //datos a variables miembro private
};

//Definición del constructor sin argumentos
Estudiante::Estudiante()
{
    Nombre[0] = 0;          //Se inicializa con el caracter nulo
    email[0] = 0;           //Se inicializa con el caracter nulo
    edad = 0;
    Expediente[0] = 0;      //Se inicializa con el caracter nulo
    Asist = 0;
    totalAsist = 0;         //Se iniciliza el total de asistencias a 0
}

//Definición del constructor con un argumento
Estudiante::Estudiante(char* p)
{
    strcpy_s(Nombre, p);    //Se inicializa con el caracter nulo
    email[0] = 0;           //Se inicializa con el caracter nulo
    edad = 0;
    Expediente[0] = 0;      //Se inicializa con el caracter nulo
    Asist = 0;
    totalAsist = 0;         //Se iniciliza el total de asistencias a 0
}

//Definición del constructor con dos argumentos
Estudiante::Estudiante(char* p, char *q)
{
    strcpy_s(Nombre, p);    //Se inicializa con el caracter nulo
    strcpy_s(email, q);     //Se inicializa con el caracter nulo
    edad = 0;
    Expediente[0] = 0; //Se inicializa con el caracter nulo
    Asist = 0;
    totalAsist = 0;         //Se iniciliza el total de asistencias a 0
}
```

```
//Definición de la función miembro de la clase Estudiante
//Se debe usar el operador de resolución de ámbito
void Estudiante::Print()

{
    printf("    Nombre: %s\n", Nombre);
    printf("     email: %s\n", email);
    printf("Expediente: %s\n", Expediente);
    printf("      Edad: %d\n", edad);
    printf("Asistencia: %d\n", Asist);
    printf("Total de asistencias: %d\n", totalAsist);
}

//Definición de la función miembro de la clase Estudiante para
//asignar datos a los miembros protected Nombre e email.
void Estudiante::Asignar(char* p, char* q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void main()
{
    Estudiante A;
    Estudiante B((char*)"Pedro");
    Estudiante C((char*)"Guadalupe",(char*)"gpe131@hotmail.com");

    A.Asignar((char*)"Pedro", (char*)"pedrogcia@yahoo.com");

    //Se asignan expedientes
    strcpy_s(A.Expediente, "00132");
    strcpy_s(B.Expediente, "00139");
    strcpy_s(C.Expediente, "00161");
    //Se asignan edades
    A.edad = 20;
    B.edad = 19;
    C.edad = 21;
    //Se asignan asistencias
    A.Asist = 1;
    B.Asist = 0;
    C.Asist = 1;
    //Se asignan total de asistencias
    A.totalAsist = 13;
    B.totalAsist = 15;
    C.totalAsist = 16;
    //Se imprimen los valores de las variables miembro
    //de cada objeto instanciado a la clase Estudiante
    A.Print(); printf("\n");
    B.Print(); printf("\n");
    C.Print(); printf("\n");
}
```

Uso del destructor en clases

La función del destructor consiste en liberar recursos del sistema, como la memoria ocupada, cerrar archivos o resguardar información relevante para su uso en posteriores ejecuciones. En la Figura 3.33 se muestra un ejemplo donde el destructor se encarga de liberar la memoria para un arreglo de datos previamente reservada por el constructor. Su implementación se detalla en el Código 3.41.

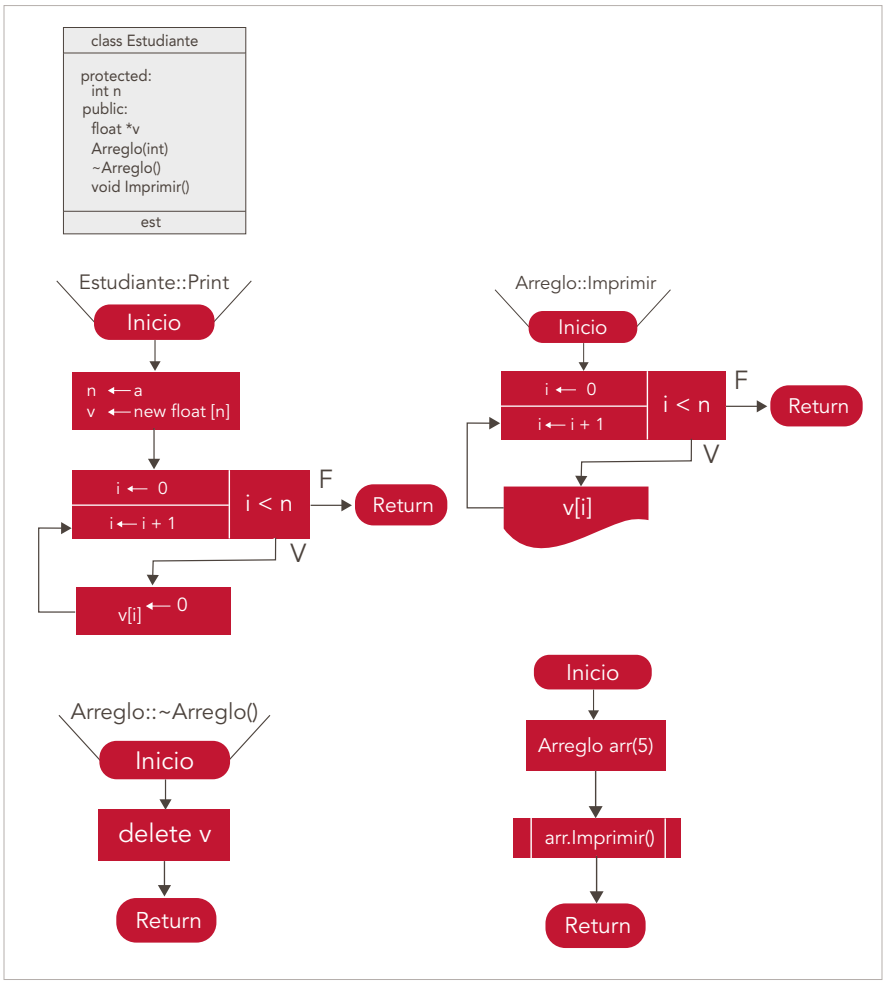


Figura 3.33. Uso del destructor para liberar memoria de un arreglo dinámico.

Código 3.41. Implementación del constructor y destructor para el manejo de memoria dinámica.

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

class Arreglo
{
protected:
    int n;
public:
    float* v;
    Arreglo(int a);
    ~Arreglo();
    void Imprimir();
};

Arreglo::Arreglo(int a)
{
    n = a;
    //Asignación de memoria dinámica a v
    v = new float[n];
    //Se inicializa los valores del arrreglo a 0
    for (int i = 0; i < n; i++)
         v[i] = 0;
}

Arreglo::~Arreglo()
{
    delete v;
}

void Arreglo::Imprimir()
{
    for (int i = 0; i < n; i++)
         cout << "v[" << i << "] = " << v[i] << endl;
}

void main()
{
    Arreglo arr(5);
    arr.Imprimir();
}
```

3.2.7. ASIGNACIÓN INICIAL DE DATOS A UNA CLASE USANDO LISTAS

Cuando se instancia un objeto de una clase, es posible proporcionar datos iniciales a sus variables miembro, tal como muestra el Código 3.42. La asignación de datos se ciñe al mismo orden de la declaración de variables. En el ejemplo mostrado se declaran `i`, `v` y `o`, a las cuales se les atribuyen los valores `{5, 3.33, 'A'}`. Notese que estos datos deben concordar con el tipo de cada variable en orden: `int`, `float` y `char`.

Código 3.42. Asignación inicial de datos a variables miembro.

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

class Ejemplo
{
public:
   int i;
   float v;
   char o;
};

void main()
{
   Ejemplo e = { 5,3.33,'A' };
   cout << "i = " << e.i << endl;
   cout << "v = " << e.v << endl;
   cout << "o = " << e.o << endl;
}
La salida es
i = 5
v = 3.33
o = A
```

Por tanto, si se modifica el orden de los valores, por ejemplo, al cambiar a `{3.33, 'A', 5}`, entonces el dato `3.33` se asignará a `i`, el carácter ASCII de `A` será establecido en `v` y el dato `5` es registrado a `o`. La salida, en consecuencia, es:

```
i = 3
v = 65
o = ♣
```

Por otro lado, si la lista está incompleta, como en el caso de `{5, 3.33}`, donde se proporcionan únicamente dos valores en lugar de tres, estos se asignarán solamente a las dos primeras variables miembro, quedando la tercera sin inicializar. En este caso, la salida será:

```
i = 5
v = 3.33
o =
```

En caso de que las variables miembro posean un especificador de acceso protegido o privado, la inicialización por lista será inaplicable.

3.2.8. ARREGLO DE CLASES

Así como hay arreglos de tipo entero, flotante o doble, también es posible crear estructuras de objetos instanciados a una clase. La clase `Ejemplo` del Código 3.43 integra tres variables miembro públicas. Al instanciarse el objeto `e`, se declara como un arreglo que cuenta con cinco objetos (`e[5]`): instancias distintas accesibles mediante un índice que las diferencia.

Código 3.43. Arreglo de objetos.

```
#include <iostream>

using namespace std;

class Ejemplo
{
public:
    int i;
    float v;
    double o;
    Ejemplo()
    {
         i = 0;
         v = 0;
         o = 300;
    }

    void Imprimir()
    {
         cout << "i=" << i << "\tv=" << v << "\to=" << o << endl;
    }
};

void main()
{
    Ejemplo e[5];
    for (int i = 0; i < 5; i++)
    {
         e[i].i = i * (i + 1) / 2;
         e[i].v = sqrt(i);
         e[i].o = i + 48;
         e[i].Imprimir();
    }
    _getch();
}
```

El resultado de la ejecución del Código 3.43 es:

```
i=0     v=0             o=48
i=1     v=1             o=49
i=3     v=1.41421       o=50
i=6     v=1.73205       o=51
i=10    v=2             o=52
```

En este caso, cada objeto, así como sus variables miembro, posee una dirección de memoria distinta; desde esta perspectiva, todo dato es independiente. A propósito, el Código 3.44 es posible utilizarse para exhibir la dirección de memoria tanto de los objetos como de las variables miembro.

Código 3.44. Direcciones de memoria asignadas a un arreglo de objetos de clase.

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

//Declaración de la clase Ejemplo
class Ejemplo
{
public:
    int i;
    float v;
    double o;
    //Constructor
    Ejemplo()
    {
        i = 0;
        v = 0;
        o = 300;
    }

    void Imprimir()
    {
        //Imprime los valores de las varaibles miembro
        cout << "i=" << i << "\tv=" << v << "\to=" << o << endl;
    }
};

void main()
{
    Ejemplo e[5];
    //Se asignan datos a las variables miembro.
    for (int i = 0; i < 5; i++)
    {
        cout << "Direcciones de memoria e[" << i << "]\n";
        cout << "e[" << i << "] = " << &e[i] << endl;
        cout << "e[" << i << "].i = " << &e[i].i << endl;
        cout << "e[" << i << "].v = " << &e[i].v << endl;
        cout << "e[" << i << "].o = " << &e[i].o << endl;
    }
}
```

Es preciso considerar que el sistema operativo asigna la memoria de manera dinámica; por consiguiente, las direcciones pueden variar en cada ejecución del programa. El resultado es el siguiente:

Direcciones de memoria `e[0]`:

```
e[0] = 002AF930   e[0].i = 002AF930 e[0].v = 002AF934 e[0].o = 002AF938
```

Direcciones de memoria `e[1]`:

```
e[1] = 002AF940   e[1].i = 002AF940 e[1].v = 002AF944 e[1].o = 002AF948
```

Direcciones de memoria `e[2]`:

```
e[2] = 002AF950   e[2].i = 002AF950 e[2].v = 002AF954 e[2].o = 002AF958
```

Direcciones de memoria `e[3]`:

```
e[3] = 002AF960   e[3].i = 002AF960 e[3].v = 002AF964 e[3].o = 002AF968
```

Direcciones de memoria `e[4]`:

```
e[4] = 002AF970   e[4].i = 002AF970 e[4].v = 002AF974 e[4].o = 002AF978
```

La Figura 3.34 representa la distribución del arreglo en la memoria. Como las variables tipo `int`, `float` y `double` ostentan un tamaño de 4 *bytes*, la separación entre cada elemento en el mapa de memoria representa ese espacio.

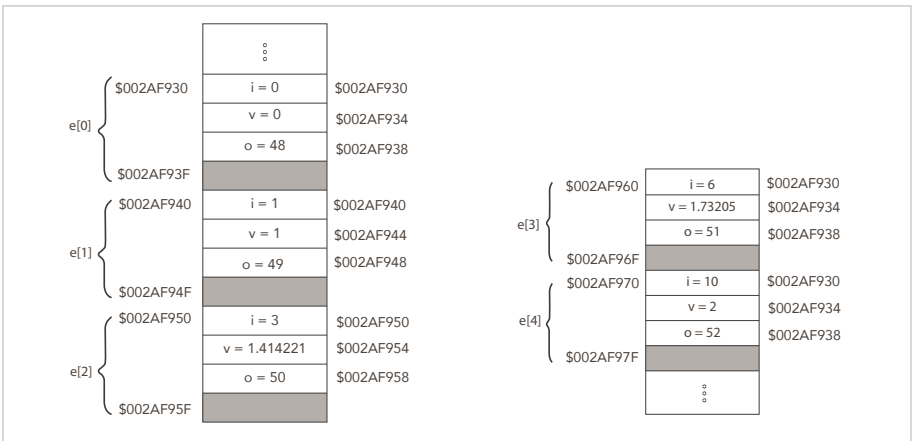


Figura 3.34. Distribucion en memoria del arreglo del objeto `e`.

3.2.9. APUNTADORES A CLASES

Los apuntadores permiten acceder a los miembros de una clase. Primero debe declararse un objeto apuntador para operarlos. A paso seguido, se debe inicializar dicha variable asignándole la dirección de memoria de otro objeto instanciado de la misma clase. El Código 3.45 ejemplifica dos formas de llevar a cabo este procedimiento.

Código 3.45. Acceso a miembros de una clase usando apuntadores.

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

//Declaración de la clase Ejemplo
class Ejemplo
{
public:
    int i;
    float v;
    double o;
    //Constructor
    Ejemplo()
    {
        i = 0;
        v = 0;
        o = 300;
    }

    void Imprimir()
    {
        //Imprime los valores de las variables miembro
        cout << "i=" << i << "\tv=" << v << "\to=" << o << endl;
    }
};

void main()
{
    Ejemplo* p;   //Se instancia un objeto como apuntador
    Ejemplo e;    //Se instancia un objeto de la misma clase

    //Se inicializa el apuntador p con la dirección del objeto e
    p = &e;
    //Primera forma de accesar a los miembros de la clase e con el apuntador p
    p->i = 15;
    p->v = 0.5;
    //Segunda forma de accesar a los miembros de la clase e con el apuntador p
    (*p).o = 1354;
    (*p).Imprimir();
}
```

Los operadores para apuntadores también se aplican a arreglos de objetos de clase, como se observa en la Figura 3.35. Como paso inicial, el apuntador `p` se orienta hacia la base del arreglo, la dirección de memoria de `e[0]`. Si `p` se incrementa en 1 (`p++`), se reorientará a la dirección de `e[1]`. En el supuesto de que se eleve en 3 (`p = p + 3`), señalará a la dirección base de `e[4]`. Un efecto similar ocurre cuando la dirección de memoria se decrementa. Por ejemplo, si el apuntador `p` está en `e[4]` y disminuye en `1` (`p--`), entonces dirigirá a la dirección base de `e[3]`. Así, es posible manipular la dirección de memoria de `p`. Cabe advertir que el programador debe abstenerse de realizar operaciones fuera de la dirección de memoria asignada para `e`.

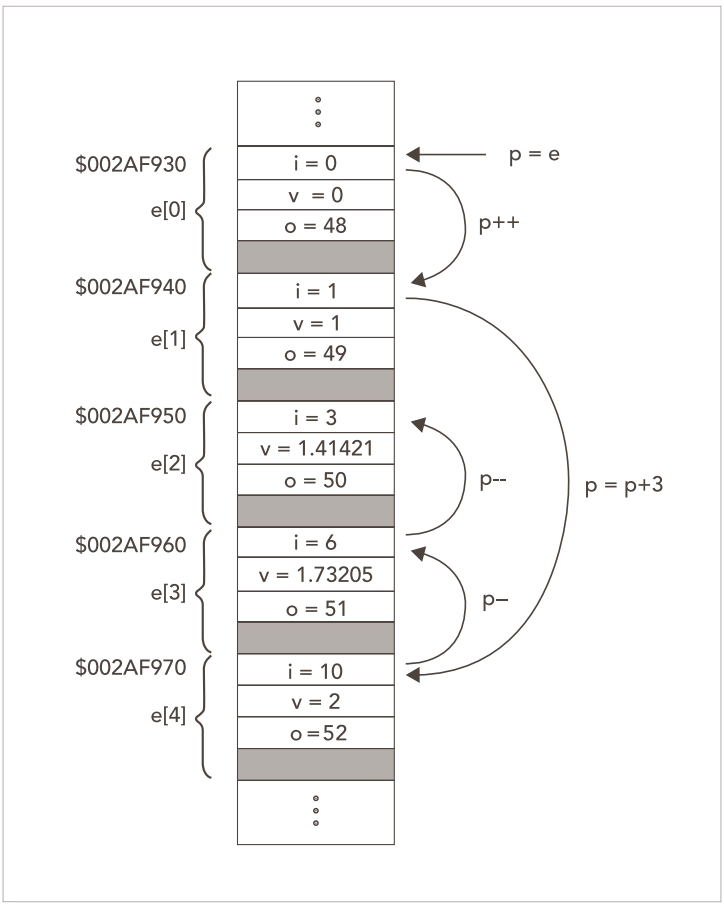


Figura 3.35. Comportamiento del apuntador al incrementar o decrementar la dirección de memoria.

El comportamiento de la Figura 3.35 se puede verificar probando el Código 3.46.

Código 3.46. Acceso a un arreglo de objetos usando un apuntador.

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

using namespace std;

//Declaración de la clase Ejemplo
class Ejemplo
{
public:
    int i;
    float v;
    double o;
    //Constructor
    Ejemplo()
    {
        i = 0;
        v = 0;
        o = 300;
    }

    void Imprimir()
    {
        //Imprime los valores de las varaibles miembro
        cout << "i=" << i << "\tv=" << v << "\to=" << o << endl;
    }
};

void main()
{
    Ejemplo* p;   //Se instancia un objeto como apuntador
    Ejemplo e[5];         //Se instancia un objeto de la misma clase

    //Se asignan datos a las variables miembro.
    cout << "Valores del arreglo\n";
    for (int i = 0; i < 5; i++)
    {
        cout << "e[" << i << "] : ";
        e[i].i = i * (i + 1) / 2;
        e[i].v = sqrt(i);
        e[i].o = i + 48;
        e[i].Imprimir();
    }

    //Se asigna al apuntador p la dirección del arreglo e
    cout << "\nPosición inicial del apuntador\n";
    p = e;
    p->Imprimir();
```

```
    cout << "Se incrementa p en 1\n";
    p++;
    p->Imprimir();
    cout << "Se incrementa p en 3\n";
    p += 3;p->Imprimir();
    cout << "Se decrementa p en 1\n";
    p--;
    p->Imprimir();
    cout << "Se decrementa p en 1\n";
    p--;
    p->Imprimir();
    _getch();
}
```

El resultado de la ejecución del código es:

```
Valores del arreglo
e[0] : i=0      v=0             o=48
e[1] : i=1      v=1             o=49
e[2] : i=3      v=1.41421       o=50
e[3] : i=6      v=1.73205       o=51
e[4] : i=10     v=2             o=52

Posición inicial del apuntador
i=0     v=0             o=48
Se incrementa p en 1
i=1     v=1             o=49
Se incrementa p en 3
i=10    v=2             o=52
Se decrementa p en 1
i=6     v=1.73205       o=51
Se decrementa p en 1
i=3     v=1.41421       o=50
```

3.2.10. EL APUNTADOR `this`

En C++, todas las clases cuentan por defecto con el apuntador **`this`**, una variable predefinida para las funciones y operadores miembro. El puntero contiene la dirección del objeto específico de la clase sobre el cual se aplica la función. Su uso es común en la programación orientada a objetos y resulta particularmente útil en contextos como la programación orientada en eventos, por ejemplo, en las aplicaciones de Windows. El Código 3.47 muestra cómo obtener la dirección del apuntador **`this`**.

Código 3.47. Dirección del apuntador `this`.

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

//Declaración de la clase Muestra
class Muestra
{
protected:
   int n;
public:
   float j;
   void* Apuntador(){return this;}
};

void main()
{
   Muestra m;
   void *p;

   //Se obtiene la dirección de m a través del this
   p = m.Apuntador();
   cout << p << "\n" << &m;
}
```

El apuntador **`this`** también permite acceder a los miembros de la clase. El siguiente código ilustra su uso.

Código 3.48. Uso de `this` para acceder a miembros de la clase.

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

//Declaración de la clase Muestra
class Muestra
{
protected:
   int n;
public:
   float j;
   Muestra(int m)
   {
        this->n = m;
        (*this).j = 3.2564;
   }
   void salida()
   {
        cout << "n = " << n << endl;
        cout << "j = " << j << endl;
```

```
    }
};

void main()
{
    Muestra m(7);
    m.salida();
}
```

3.2.11. CLASES DERIVADAS

También conocida como herencia de clases, tal noción proviene de una metáfora biológica, y describe cómo un objeto puede reutilizar y ampliar las características de otro sin necesidad de reescribirlas desde cero. Constituye una propiedad esencial de la POO, puesto que posibilita la creación nuevas clases derivadas de otras existentes, conservando las variables y funciones miembro de la clase original, además de incorporar nuevos elementos. La Figura 3.36 ilustra la noción.

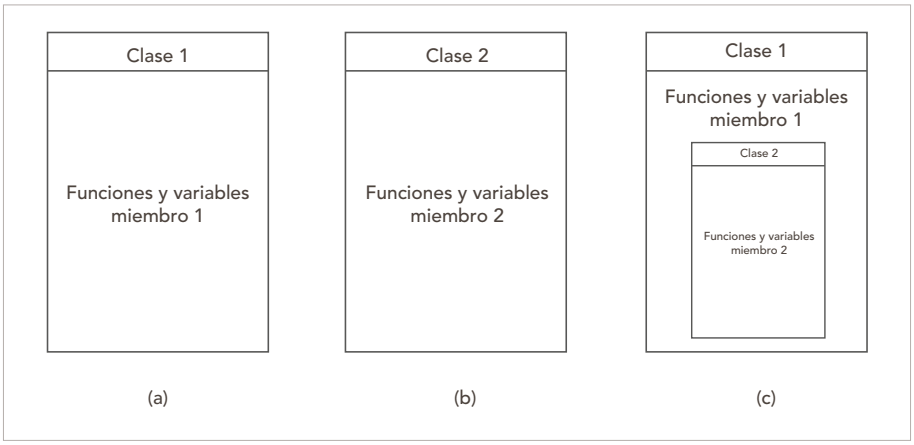


Figura 3.36. Herencia de clases. (a) Clase derivada. (b) Clase base. (c) La clase 1 hereda a la clase 2.

Al momento de desplegar la herencia de clases, se establece una relación entre una clase base y una derivada. Los integrantes (variables y funciones) de la clase base son heredados por la clase derivada. De esta manera, se extienden las características de la clase original con nuevos elementos. La sintaxis correspondiente se ilustra en el Código 3.49.

Código 3.49. Sintaxis para obtener una clase derivada.

```
//Se define la clase base
class [<identificador_clase_base>]
{
<especificador_acceso:>
    <tipo> <nombre_variable(s)>;
    <tipo> <nombre_variable(s)>;
    <tipo> <definición de función o función prototipo>;
    …
;

//Se define la clase derivada
class [<identificador_clase>] : <especificador_acceso> <identificador_clase_base>
{
<especificador_acceso:>
    <tipo> <nombre_variable(s)>;
    <tipo> <nombre_variable(s)>;
    <tipo> <definición de función o función prototipo>;
    …
};
```

Cabe destacar que existe una diferencia importante respecto a los especificadores de acceso, como se muestra en la Figura 3.37.

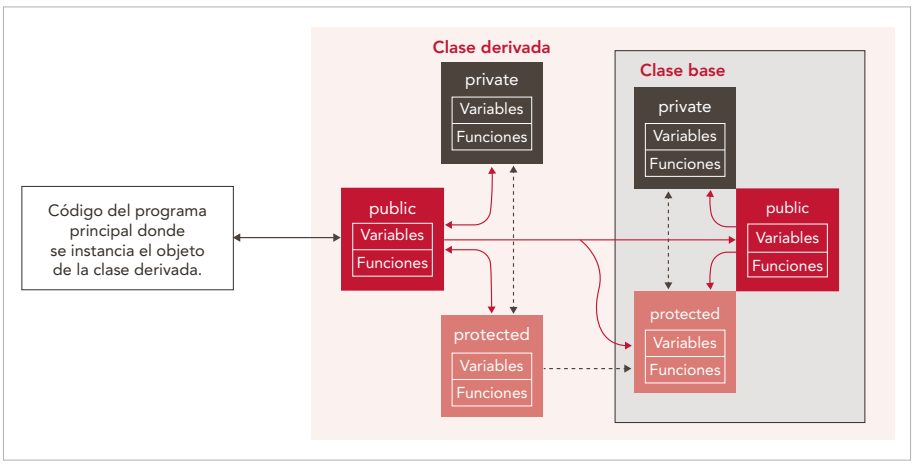


Figura 3.37. Funcionamiento de los especificadores de acceso en una clase derivada.

Con anterioridad se expusieron tres niveles de miembros en una clase: públicos (**`public`**), protegidos (**`protected`**) y privados (**`private`**). Cada tipo conlleva una forma específica de acceso. Los miembros declarados como **`public`** son accesibles desde cualquier parte del código; en el caso de **`protected`**, el acceso se limita a las clases derivadas o procedentes de los

miembros protegidos de la clase base. Por último, en **`private`** solamente son accesibles las funciones de la misma clase, lo que significa que los miembros privados de la clase derivada pueden utilizarse solo dentro de esa clase; a su vez, los de la clase base son accesibles desde las funciones que pertenecen a ella. El Código 3.50 ejemplifica, con la clase base (**`A`**) y la clase derivada (**`B`**), cómo se lleva a cabo el uso de los diferentes especificadores de acceso.

Código 3.50. Ejemplo de uso de especificadores de acceso.

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

using namespace std;

class A
{
private:
    int i;
protected:
    float a;
public:
    char o;
    void Asignar()
    {
         //Un miembro private solamente puede ser accesado desde
         //una función miembro de la misma clase
         i = 5;
    }
};

class B : public A
{
protected:
    int j;
public:
    B()
    {
         j = 0;  //Puede ser accesado pues es público en Derivada
         o = 's';//Puede ser accesado pues es público en Base
         a = 0.5;//Puede ser accesado aunque es protected en Base
         i = 4; //No es válida, pues no se puede accesar al miembro
    }
    void Lecturas() {}
};

void main()
{
    B d;

    d.j = -3; //No se puede accesar desde main debido a que es protected
    d.o = 'm';//Se puede accesar desde main pues es público
    d.a = 1.3;//No se puede accesar desde main pues es private en Base
    d.i = 10; //No se puede accesar desde main
}
```

El Código 3.51 puntualiza la interacción de los especificadores de acceso en una clase derivada. En lo que concierne a la omisión de especificadores de acceso, entonces se consideran **`private`** por defecto.

Código 3.51. Ejemplo de la interacción de miembros de una clase derivada.

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

//Declaración de la clase Base
class Base
{
private:
    int A;
    void PrivadoB();
protected:
    double C;
    void ProtegidoB();
public:
    float E;
    void PublicoB();
};

void Base::PrivadoB()
{
    //Desde la función privada se accesa a miembros protected y public
    A = 3;         //Miembro private
    C = 5;         //Miembro protected
    E = 12.456; //Miembro public
}

void Base::ProtegidoB()
{
    //Desde la función protegida se accesa a miembros private y public
    A = 6;         //Miembro private
    C = 8;         //Miembro protected
    E = 10.658; //Miembro public
}

void Base::PublicoB()
{
    //Desde la función public se accesa a miembros private y protected
    A = -1;        //Miembro private
    C = -4;        //Miembro protected
    E = 3.1416; //Miembro public
}

class Derivada : public Base
{
private:
    int B;
    void PrivadoD();
protected:
    double D;
    void ProtegidoD();
```

```
public:
    float F;
    void PublicoD();
};

void Derivada::PrivadoD()
{
    //Accesando a miembros de la clase base desde la clase derivada
    A = 34;        //Miembro private de la base no puede ser accesado
    C = -27;       //Miembro protected puede ser accesado
    E = 15.608;    //Miembro public puede ser accesado
}

void Derivada::ProtegidoD()
{
    //Accesando a miembros de la clase base desde la clase derivada
    A = -27;       //Miembro private de la base no puede ser accesado
    C = 15;        //Miembro protected puede ser accesado
    E = -3.271;    //Miembro public puede ser accesado
}

void Derivada::PublicoD()
{
    //Accesando a miembros de la clase base desde la clase derivada
    A = 100;       //Miembro private de la base no puede ser accesado
    C = -48;       //Miembro protected puede ser accesado
    E = -1.359;    //Miembro public puede ser accesado
}

void main()
{
    Derivada b;
    //La clase Derivada hereda los miembros de la clase Base
    //No todos los miembros pueden ser accesados desde main

    //Accesando a las funciones miembro de la clase derivada desde el main
    b.PrivadoD();           //No se puede accesar a un miembro private
    b.ProtegidoD();         //No se puede accesar a un miembro protected
    b.PublicoD();           //SI se puede accesar a un miembro public

    //Accesando a las funciones miembro de la clase base desde el main
    b.PrivadoB();           //No se puede accesar a un miembro private
    b.ProtegidoB();         //No se puede accesar a un miembro protected
    b.PublicoB();           //SI se puede accesar a un miembro public
}
```

Aplicando herencia de clases

El siguiente ejemplo administra a la clase base **`Persona`** los atributos **`Nombre`**, **`email`** y **`Edad`**. En ella se ejecutan tres funciones: el constructor entrega sus valores de inicio a las variables miembro; **`Imprimir()`** presenta su contenido, y **`Asignar()`** posibilita su modificación. La Figura 3.38 y el Código 3.52 exhiben la definición de las clases **`Persona`** y **`Estudiante`**.

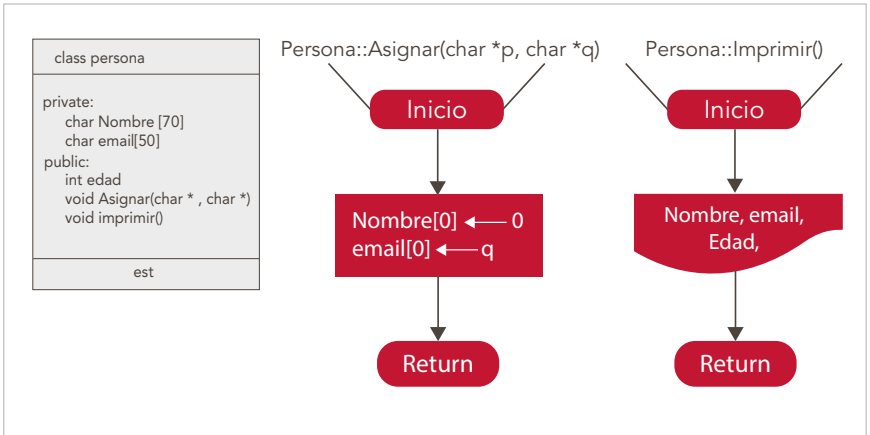


Figura 3.38. Clase base `Persona`.

Código 3.52. Retomando la clase `Estudiante`.

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

using namespace std;

class Persona
{
private:
    char Nombre[70];
    char email[50];
public:
    int Edad;
    void Asignar(char* p, char* q);
    void Imprimir();
};

void Persona::Asignar(char* p, char* q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void Persona::Imprimir()
{
    printf("Nombre: %s\n", Nombre);
    printf(" email: %s\n", email);
    printf("  edad: %d\n", Edad);
}
```

Acto seguido, se declara la clase **Estudiante**, la cual despliega la información guardada en **Persona**. La nueva información que se incorpora es **Dirección**, **Teléfono** y **Expediente**. La Figura 3.39 muestra la clase derivada **Estudiante**; por su parte, el Código 3.53 presenta el programa completo.

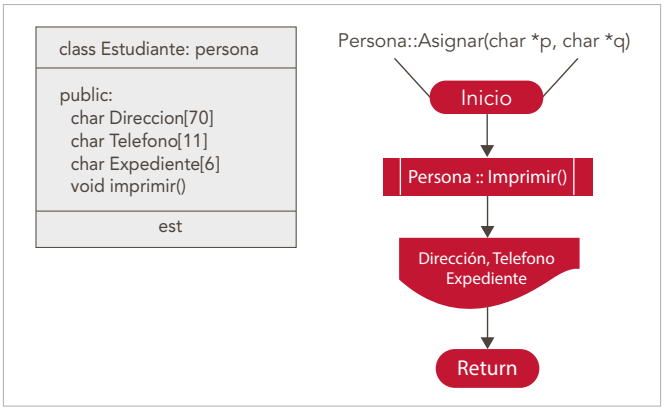


Figura 3.39. Clase Estudiante derivada de Persona.

Código 3.53. La clase Estudiante hereda a la clase Persona.

```
//Estudiante hereda a Persona
class Estudiante : public Persona
{
public:
   char Direccion[70];
   char Telefono[11];
   char Expediente[6];
   void Imprimir();
};

void Estudiante::Imprimir()
{
   Persona::Imprimir();
   printf(" Dirección: %s\n", Direccion);
   printf("  Teléfono: %s\n", Telefono);
   printf("Expediente: %s\n", Expediente);
}
```

Para trabajar con la clase derivada **Estudiante** desde **main()**, es preciso realizar el programa conforme a la Figura 3.40 y el Código 3.54.

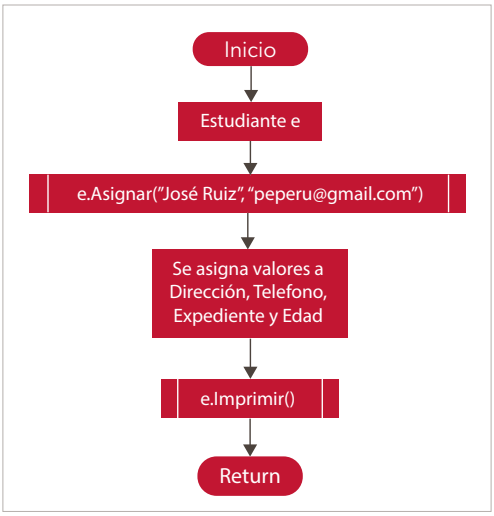


Figura 3.40. Uso de la clase derivada `Estudiante`.

Código 3.54. Código para usar la clase derivada `Estudiante`.

```
void main()
{
    //se intancia el objeto a la clase derivada
    Estudiante e;
    //Se asigna valores a las variables miembro
    e.Asignar((char*)"José Ruiz", (char*)"peperu@gmail.com");
    strcpy_s(e.Direccion, "Calle del reprobado 403");
    strcpy_s(e.Telefono, "4424423333");
    strcpy_s(e.Expediente, "00301");
    e.Edad = 19;
    //Se manda imprimir los valores
    e.Imprimir();
    _getch();
}
```

El Código 3.55 muestra el programa completo.

Código 3.55. Programa completo para la clase derivada `Estudiante`.

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

using namespace std;
```

```
//Se define la clase base
class Persona
{
private:
    char Nombre[70];
    char email[50];
public:
    int Edad;
    void Asignar(char* p, char* q);
    void Imprimir();
};

void Persona::Asignar(char* p, char* q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void Persona::Imprimir()
{
    printf("Nombre: %s\n", Nombre);
    printf(" email: %s\n", email);
    printf("  edad: %d\n", Edad);
}

//Se define la clase derivada
//Estudiante hereda a Persona
class Estudiante : public Persona
{
public:
    char Direccion[70];
    char Telefono[11];
    char Expediente[6];
    void Imprimir();
};

void Estudiante::Imprimir()
{
    Persona::Imprimir();
    printf(" Dirección: %s\n", Direccion);
    printf("  Telefono: %s\n", Telefono);
    printf("Expediente: %s\n", Expediente);
}

void main()
{
    //se intancia el objeto a la clase derivada
    Estudiante e;
    //Se asigna valores a las variables miembro
    e.Asignar((char*)"José Ruiz", (char*)"peperu@gmail.com");
    strcpy_s(e.Direccion, "Calle del reprobado 403");
    strcpy_s(e.Telefono, "4424423333");
    strcpy_s(e.Expediente, "00301");
    e.Edad = 19;
    //Se manda imprimir los valores
    e.Imprimir();
    _getch();
}
```

La salida del programa es:

```
Nombre: José Ruiz
email: peperu@gmail.com
Edad: 19
Dirección: Calle del reprobado 403
Telefono: 4424423333
Expediente: 00301
```

Uso de constructores en clases derivadas o heredadas

En una relación de herencia, es viable aplicar constructores del mismo modo que se hace en clases sin herencia. En tales casos, se ejecutan de forma encadenada; primero el constructor de la clase base y, después, el de la clase derivada. En el diagrama de flujo de la Figura 3.41 se muestra el proceso en que la clase base exhibe el identificador **`Persona`** y la derivada utiliza el identificador **`Estudiante`**.

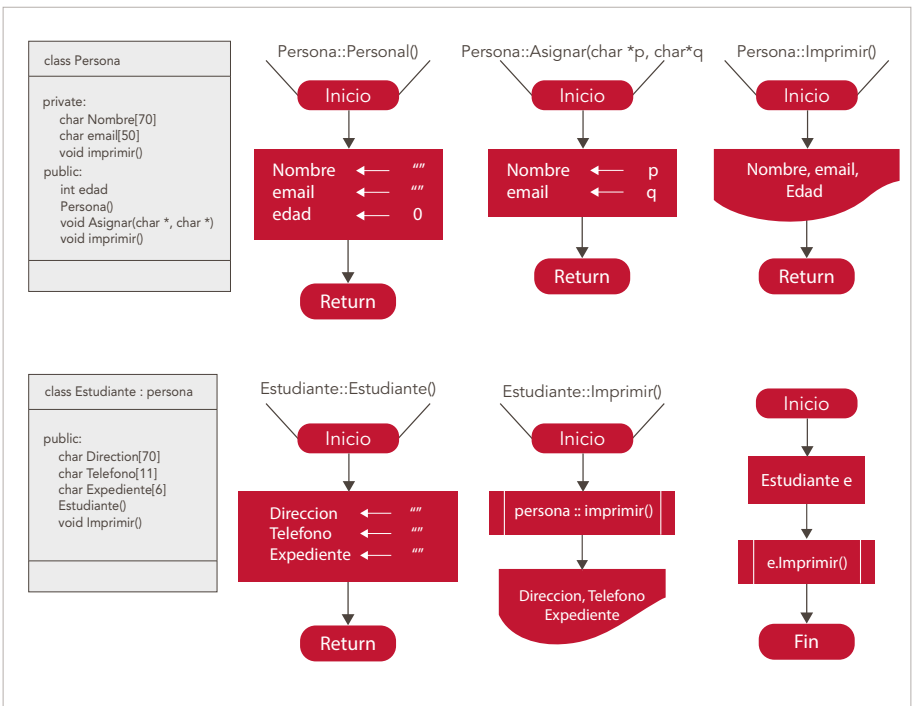


Figura 3.41. Constructores en clases heredadas.

Cada clase dispone de su propio constructor. Al instanciar el objeto `e` a partir de la clase derivada, primero se ejecuta el constructor de la clase base y, enseguida, el de la clase derivada. Por consiguiente, todas las variables quedan inicializadas. Al acabar, el programa imprime los valores resultantes. El Código 3.56 muestra la implementación correspondiente.

Código 3.56. Uso de constructores en herencia de clases.

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

//Se define la clase base
class Persona
{
private:
    char Nombre[70];
    char email[50];
public:
    int Edad;
    Persona(); //Prototipo del constructor
    void Asignar(char* p, char* q);
    void Imprimir();
};

//Definición del constructor de la clase base
Persona::Persona()
{
    Nombre[0] = 0;
    email[0] = 0;
    Edad = 0;
}

void Persona::Asignar(char* p, char* q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void Persona::Imprimir()
{
    printf("Nombre: %s\n", Nombre);
    printf(" email: %s\n", email);
    printf("  edad: %d\n", Edad);
}

//Se define la clase derivada
//Estudiante hereda a Persona
class Estudiante : public Persona
{
public:
    char Direccion[70];
    char Telefono[11];
    char Expediente[6];
```

```
    Estudiante();           //Prototipo del constructor
    void Imprimir();
};

//Se define el constructor de la clase derivada
Estudiante::Estudiante()
{
    Direccion[0] = 0;
    Telefono[0] = 0;
    Expediente[0] = 0;
}

void Estudiante::Imprimir()
{
    Persona::Imprimir();

    printf(" Dirección: %s\n", Direccion);
    printf("  Telefono: %s\n", Telefono);
    printf("Expediente: %s\n", Expediente);
}

void main()
{
    //se instancia el objeto a la clase derivada
    Estudiante e;

    //Se manda imprimir los valores
    e.Imprimir();
}
```

La salida de Código 3.56 evidencia que, al instanciar el objeto, ante todo se ejecuta el constructor de la clase base y, a continuación, el de la clase derivada.

```
Se ejecuta el constructor de la clase base
Se ejecuta el constructor de la clase base
Nombre:
email:
Edad: 0
Dirección:
Teléfono:
Expediente:
```

En el caso de clases derivadas secuenciales, sus constructores se ejecutan en el mismo orden que las rige, uno por una desde la base hasta la última clase derivada en la secuencia. En el Código 3.57 puede observarse el proceso, al respecto, hay que advertir que es indispensable desginar los constructores de `A` y `B` como públicos, de lo contrario, no podrán invocarse.

Código 3.57. Secuencia de ejecución del constructor en una estructura de clases derivadas.

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

//Se define la clase A
class A
{
public:
    A() //Prototipo del constructor A
    {
         cout << "Se ejecuta el constructor de la clase A\n";
    }
};

//Se define la clase B
class B : public A
{
public:
    B() //Prototipo del constructor B
    {
         cout << "Se ejecuta el constructor de la clase B\n";
    }
};

//Se define la clase C
class C : public B
{
public:
    C() //Prototipo del constructor C
    {
         cout << "Se ejecuta el constructor de la clase C\n";
    }
};

void main()
{
    //se instancia el objeto a la clase derivada
    C c;
}
```

La salida del Código 3.57 muestra que el constructor de la clase **`A`** se ejecuta en primer lugar, seguido por el de la clase **`B`** y, al final, por el de la clase **`C`**.

```
Se ejecuta el constructor de la clase A
Se ejecuta el constructor de la clase B
Se ejecuta el constructor de la clase C
```

Ahora bien, es habitual enviar datos a los constructores. En el Código 3.58 se aprecia que el constructor de la clase base **`Persona`** no recibe argumentos, mientras que el constructor de la clase derivada **`Estudiante`** requiere un argumento para su inicialización.

Código 3.58. Constructor de la clase derivada con argumentos.

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

//Se define la clase base
class Persona
{
private:
    char Nombre[70];
    char email[50];
public:
    int Edad;
    Persona(); //Prototipo del constructor sin argumentos
    void Asignar(char* p, char *q);
    void Imprimir();
};

//Definición del constructor de la clase base Persona
Persona::Persona()
{
    cout << "Se asigna el nombre en Persona\n";
    Nombre[0] = 0;
    email[0] = 0;
    Edad = 0;
}

void Persona::Asignar(char* p, char *q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void Persona::Imprimir()
{
    printf("Nombre: %s\n", Nombre);
    printf(" email: %s\n", email);
    printf("  edad: %d\n", Edad);
}

//Se define la clase derivada
//Estudiante hereda a Persona
class Estudiante : public Persona
{
public:
    char Direccion[70];
    char Telefono[11];
    char Expediente[6];
    Estudiante(char* p);    //Prototipo del constructor con argumento
    void Imprimir();
};

//Definición del prototipo del constructor de la clase derivada con argumento
Estudiante::Estudiante(char* p)
{
    cout << "Se asigna la dirección en Estudiante\n";
    strcpy_s(Direccion, p);
```

```
    Telefono[0] = 0;
    Expediente[0] = 0;
}

void Estudiante::Imprimir()
{
    Persona::Imprimir();
    printf(" Dirección: %s\n", Direccion);
    printf("  Telefono: %s\n", Telefono);
    printf("Expediente: %s\n", Expediente);
}

void main()
{
    //se instancia el objeto a la clase derivada
    Estudiante e((char*)"Calle del reprobado");
    //Se manda imprimir los valores
    e.Imprimir();
    _getch();
}
```

La ejecución del Código 3.58 produce la siguiente salida:

```
Se asigna el nombre en Persona
Se asigna la dirección en Estudiante
Nombre:
email:
Edad: 0
Dirección: Calle del reprobado
Teléfono:
Expediente:
```

Sin embargo, en numerosas ocasiones es necesario que el constructor de la clase base también reciba argumentos. Dentro del Código 3.59 se distingue cómo el constructor de la clase derivada **`Estudiante`** admite dos argumentos, de los cuales uno se consigna directamente al constructor de la clase base **`Persona`**.

Código 3.59. Transferencia de argumentos al constructor de la clase base.

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

//Se define la clase base
class Persona
{
private:
    char Nombre[70];
    char email[50];
public:
```

```
    int Edad;
    Persona(char *p); //Prototipo del constructor con 1 argumento
    void Asignar(char* p, char *q);
    void Imprimir();
};

//Definición del constructor de la clase base Persona que recibe 1 argumento
//El argumento es una cadena de caracteres que va a ser asignada a Nombre
Persona::Persona(char *p)
{
    cout << "Se asigna el nombre en Persona\n";
    strcpy_s(Nombre, p);
    email[0] = 0;
    Edad = 0;
}

void Persona::Asignar(char* p, char *q)
{
    strcpy_s(Nombre, p);
    strcpy_s(email, q);
}

void Persona::Imprimir()
{
    printf("Nombre: %s\n", Nombre);
    printf("email: %s\n", email);
    printf("Edad: %d\n", Edad);
}

//Se define la clase derivada
//Estudiante hereda a Persona
class Estudiante : public Persona
{
public:
    char Direccion[70];
    char Telefono[11];
    char Expediente[6];
    Estudiante(char* p, char* q);    //Prototipo del constructor
    void Imprimir();
};

//Definición del constructor de la clase derivada Estudiante, que recibe dos
//argumentos, y uno de ellos es enviado al constructor de la clase derivada
//Persona, que es la variable q.
Estudiante::Estudiante(char* p, char* q) : Persona(q)
{
    cout << "Se asigna la dirección en Estudiante\n";
    strcpy_s(Direccion, p);
    Telefono[0] = 0;
    Expediente[0] = 0;
}

void Estudiante::Imprimir()
{
    Persona::Imprimir();
    printf("Dirección: %s\n", Direccion);
    printf("Teléfono: %s\n", Telefono);
```

```
    printf("Expediente: %s\n", Expediente);
}

void main()
{
    //Se instancia el objeto a la clase derivada
    Estudiante e((char*)"Calle del reprobado", (char*)"José");
    //Se manda imprimir los valores
    e.Imprimir();
}
```

El resultado de la ejecución del programa es el siguiente:

```
Se asigna el nombre en Persona
Se asigna la dirección en Estudiante
Nombre: José
email:
Edad: 0
Dirección: Calle del reprobado
Teléfono:
Expediente:
```

3.2.12. POLIMORFISMO Y FUNCIONES VIRTUALES

El polimorfismo de C++ permite que un objeto sea declarado como un tipo, pero se comporte como otro. En el Código 3.60 puede verse este mecanismo en acción; se define la clase base `Ejemplo` y dos clases derivadas, `A` y `B`. En `main()`, el apuntador `e` se declara como tipo `Ejemplo`, pero se reserva memoria a un objeto de tipo `A`. Un efecto semejante ocurre con el apuntador `f`.

Código 3.60. Uso de polimorfismo en clases.

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

//Se define la clase base Ejemplo
class Ejemplo
{
public:
    void Imprimir(){cout << "El valor es 5\n";}
};

//Se define la clase base A, derivada de Ejemplo
class A : public Ejemplo
{
```

```
public:
    void Imprimir() { cout << "El valor es 10\n"; }
};

//Se define la clase base B, derivada de Ejemplo
class B : public Ejemplo
{
public:
    void Imprimir() {
         cout << "El valor es 15\n";
    }
};

void main()
{
    Ejemplo* e = new A();//Se declara el objeto e tipo Ejemplo y A
    Ejemplo* f = new B();//Se declara el objeto f tipo Ejemplo y B

    e->Imprimir(); //Se ejecuta la función de la clase A
    f->Imprimir(); //Se ejecuta la función de la clase B
}
```

El resultado de la ejecución es:

```
El valor es 5
El valor es 5
```

Dicho de otro modo, se ejecuta la función definida de la clase base. La razón es que el objeto se declara como tipo **`Ejemplo`**, por ende, se invoca la función asociada a dicha clase. En caso de desear que, en su lugar, se ejecuten las funciones de las clases derivadas **`A`** y **`B`**, es menester declarar como virtual la función **`Imprimir()`** de la clase **`Ejemplo`**, tal como se muestra en el Código 3.61.

Código 3.61. Función virtual en la clase base `Ejemplo`.

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

//Se define la clase base Ejemplo
class Ejemplo
{
public:
    void virtual Imprimir(){cout << "El valor es 5\n";}
};

//Se define la clase base A, derivada de Ejemplo
class A : public Ejemplo
{
```

```
public:
    void Imprimir() { cout << "El valor es 10\n"; }
};

//Se define la clase base B, derivada de Ejemplo
class B : public Ejemplo
{
public:
    void Imprimir() {
         cout << "El valor es 15\n";
    }
};

void main()
{
    Ejemplo* e = new A();//Se declara el objeto e tipo Ejemplo y A
    Ejemplo* f = new B();//Se declara el objeto f tipo Ejemplo y B

    e->Imprimir(); //Se ejecuta la función de la clase A
    f->Imprimir(); //Se ejecuta la función de la clase B
}
```

Al ejecutar esta función el resultado es:

```
El valor es 10
El valor es 15
```

Por ello, en esta ejecución se activan las funciones correspondientes de las clases **`A`** y **`B`**. Las funciones virtuales se declaran en la clase base para posibilitar su redefinición en las clases derivadas, lo que garantiza que se invoque la versión adecuada en cada caso. Si se requiere llamar a la función **`Imprimir()`** de la clase base, debe hacerse de la forma indicada en **`main()`** del Código 3.62.

Código 3.62. Se manda llamar a la función virtual de `Ejemplo` en `main()`.

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

//Se define la clase base Ejemplo
class Ejemplo
{
public:
    void virtual Imprimir(){cout << "El valor es 5\n";}
};

//Se define la clase base A, derivada de Ejemplo
class A : public Ejemplo
{
```

```
public:
    void Imprimir() { cout << "El valor es 10\n"; }
};

//Se define la clase base B, derivada de Ejemplo
class B : public Ejemplo
{
public:
    void Imprimir() {
         cout << "El valor es 15\n";
    }
};

void main()
{
    Ejemplo* e = new A();//Se declara el objeto e tipo Ejemplo y A
    Ejemplo* f = new B();//Se declara el objeto f tipo Ejemplo y B

    e->Imprimir();                  //Se ejecuta la función de la clase A
    f->Imprimir();                  //Se ejecuta la función de la clase B
    e->Ejemplo::Imprimir();//Se ejecuta la función de la clase base Ejemplo
}
```

3.2.13. SOBRECARGA DE OPERADORES PARA CLASES

La sobrecarga de operadores se implementa de forma similar a como se realiza en las estructuras. En el Código 3.63 se ilustra el procedimiento para sobrecargar algunos operadores.

Código 3.63. Sobrecarga de operadores para clases.

```
#include <iostream>
#include <conio.h>
using namespace std;

class Complejo
{
public:
    float r, i;
    void Print()
    {
         cout << r << " + " << i << "i" << endl;
    }
};

// Sobrecarga del operador + para Complejo
Complejo operator +(Complejo a, Complejo b) {
    Complejo res = { a.r + b.r, a.i + b.i };
    return res;
}

// Sobrecarga del operador - para Complejo
Complejo operator -(Complejo a, Complejo b) {
```

```
    Complejo res = { a.r - b.r, a.i - b.i };
    return res;
}

// Sobrecarga del operador ++ para Complejo
Complejo operator ++(Complejo& a)
{
    Complejo* p, res;
    p = &a;
    p->r++;
    p->i++;
    res = { a.r, a.i };
    return res;
}

// Sobrecarga del operador -- para Complejo
Complejo operator --(Complejo& a)
{
    Complejo* p, res;
    p = &a;
    p->r--;
    p->i--;
    res = { a.r, a.i };
    return res;
}

int main()
{
    Complejo A = { 2, 3 };
    Complejo B = { -1, 4 };
    Complejo S, R;
    cout << "A = "; A.Print();
    cout << "B = "; B.Print();
    S = A + B;
    R = A - B;
    ++A;
    --B;
    cout << "S = A + B = "; S.Print();
    cout << "R = A - B = "; R.Print();
    cout << "++A = "; A.Print();
    cout << "--B = "; B.Print();
    _getch();
    return 1;
}
```

Se obtiene la siguiente salida:

```
A = 2 + 3i
B = -1 + 4i
S = A + B = 1 + 7i
R = A - B = 3 + -1i
++A = 3 + 4i
--B = -2 + 3i
```

Asimismo, es factible declarar la sobrecarga dentro de la clase, como se ilustra en el Código 3.64. Una de las ventajas de adoptar este enfoque es que habilita la sobrecarga del operador de asignación `=`.

Código 3.64. Sobrecarga de operadores declarados en la clase.

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

class Complejo
{
public:
    float r, i;
    Complejo();
    Complejo(float a, float b);
    Complejo& operator+ (Complejo &a);
    Complejo& operator- (Complejo &a);
    Complejo& operator++ ();
    Complejo& operator-- (int a);
    Complejo& operator= (Complejo &a);
    void Print()
    {
         cout << r << " + " << i << "i" << endl;
    }
};

Complejo::Complejo()
{
    r = 0.0;
    i = 0.0;
}

Complejo::Complejo(float a, float b)
{
    r = a;
    i = b;
}

// Sobrecarga del operador + para Complejo
Complejo& Complejo::operator+ (Complejo &a)
{
    this->r += a.r;
    this->i += a.i;
    return *this;
}

// Sobrecarga del operador - para Complejo
Complejo& Complejo::operator- (Complejo &a) {
    this->r -= a.r;
    this->i -= a.i;
    return *this;
}
```

```
// Sobrecarga del operador ++ para Complejo
Complejo& Complejo::operator++ ()
{
    this->r++;
    this->i++;
    return *this;
}

// Sobrecarga del operador -- para Complejo
Complejo& Complejo::operator-- (int a)
{
    this->r--;
    this->i--;
    return *this;
}

Complejo& Complejo::operator= (Complejo &a)
{
    this->r = a.r;
    this->i = a.i;
    return *this;
}

int main()
{
    Complejo A = { 2, 3 };
    Complejo B = { -1, 4 };
    Complejo S;

    cout << "A = "; A.Print();
    cout << "B = "; B.Print();
    cout << "A + B" << endl;
    A + B;
    cout << "A = "; A.Print();
    cout << "B = "; B.Print();
    cout << "A - B" << endl;
    A - B;
    cout << "A = "; A.Print();
    cout << "B = "; B.Print();
    cout << "++A" << endl;
    ++A;
    cout << "A = "; A.Print();
    cout << "B--" << endl;
    B--;
    cout << "B = "; B.Print();
    S = A;
    cout << "S = "; S.Print();
    return 1;
}
```

El resultado de la ejecución del código es:

```
A = 2 + 3i
B = -1 + 4i
A + B
```

```
A = 1 + 7i
B = -1 + 4i
A - B
A = 2 + 3i
B = -1 + 4i
++A
A = 3 + 4i
B--
B = -2 + 3i
S = 3 + 4i
```

El Código 3.65 presenta un ejemplo de sobrecarga del operador = aplicado a cadenas. En él puede apreciarse el uso del apuntador **this**, empleado para acceder a los datos miembros de la clase y efectuar las operaciones correspondientes.

Código 3.65. Sobrecarga de operadores aplicada a cadenas.

```
#include <iostream>
#include <conio.h>
using namespace std;

class Cadenas
{
private:
    int len;       //Longitud de la cadena
public:
    char* cadena;
    Cadenas();
    Cadenas(char* p);
    Cadenas& operator= (Cadenas& a);
};

Cadenas::Cadenas()
{
    cadena = new char[1];
    cadena[0] = 0;
    len = 1;
}

Cadenas::Cadenas(char* p)
{
    cadena = new char[strlen(p) + 1];
    strcpy_s(cadena, strlen(cadena), p);
    len = strlen(cadena);
}

Cadenas& Cadenas::operator= (Cadenas& a)
{
    if (this != &a)
    {
         delete[] cadena;
         cadena = new  char[strlen(a.cadena) + 1];
         strcpy_s(cadena, strlen(cadena), a.cadena);
```

```
    }
    return *this;
}

void main()
{
    Cadenas C1;
    Cadenas C2((char*)"Hola");

    cout << "C1 = " << C1.cadena << endl;
    cout << "C2 = " << C2.cadena << endl;
    C1 = C2;
    cout << "C1 = " << C1.cadena << endl;
    _getch();
}
```

El resultado de la ejecución del programa es:

```
C1 =
C2 = Hola
C1 = Hola
```

El Código 3.66 muestra la sobrecarga del operador **+** para la clase **`Cadena`**. Aquí también se recurre al apuntador **`this`** para acceder a las variables miembro de la propia clase.

Código 3.66. Sobrecarga del operador + para cadenas.

```
#include <iostream>
#include <string.h>
#include <conio.h>
using namespace std;

class Cadena
{
public:
    int n; //Longitud de la cadena
    char S[50];

    char* p;
    Cadena(char* q)
    {
         strcpy_s(S, q);
         //p = q;
    }
    Cadena()
    {
         S[0] = 0;
         //*p = 0;
    }
};
```

```
/* Definición del operador + para complejos */
Cadena operator +(Cadena a, Cadena b) {
    Cadena temp;
    strcpy_s(temp.S, a.S);
    strcat_s(temp.S, b.S);
    return temp;
}

void main()
{
    Cadena a((char*)"Hola");
    Cadena b((char*)"clase");
    Cadena e((char*)" ");

    Cadena s;

    cout << "a = " << a.S << endl;
    cout << "b = " << b.S << endl;

    s = a + e + b;
    cout << "s = " << s.S << endl;
    _getch();
}
```

El resultado corresponde a:

```
a = Hola
b = clase
s = Hola clase
```

Ejercicios: Sobrecarga de operadores

1. Implemente un programa que calcule áreas y permita efectuar operaciones de suma y resta entre ellas. Para el área del triángulo, utilice la fórmula correspondiente. Aplique sobrecarga de operadores.
2. Realice un programa que permita llevar a cabo operaciones de concatenación entre cadenas mediante la sobrecarga de operadores.

3.2.14. OBJETOS PLANTILLA

Una plantilla o *template* puede concebirse como un molde reutilizable en diferentes situaciones y con diversos propósitos. Su uso resulta conveniente cuando el programa adquiere

complejidad y aparecen segmentos de código que, si bien lucen similares entre sí, contienen en sus líneas variaciones puntuales que los distinguen de manera significativa. La plantilla es aplicable tanto a funciones como a clases.

Funciones plantilla

Para explicar la utilidad de las funciones plantilla, se toma como referencia el Código 3.67. En tal caso, la función **`Array()`** genera un arreglo de enteros y, posteriormente, sus elementos se imprimen en **`main`**. Aun cuando el código es sencillo, si se requiere realizar la misma operación con un arreglo de tipo flotante, debe crearse una segunda función, tal como se expone más adelante en el Código 3.68. Como puede observarse, ambas funciones son prácticamente idénticas.

Código 3.67. Generación de un arreglo dinámico.

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

using namespace std;

int* Array(int n)
{
    int* p = new int[n];
    for (int i = 0; i < n; i++)
        p[i] = i;
    return p;
}

void main()
{
    int n = 10;
    int* q;

    //Se genera el arreglo en la función Array
    q = Array(n);
    //Se imprime el arreglo generado
    for (int i = 0; i < n; i++)
        cout << q[i] << "\t";
    _getch();
}
```

Código 3.68. Creación de dos arreglos dinámicos de tipo diferente.

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

using namespace std;

int* Array(int n)
{
    int* p = new int[n];
    for (int i = 0; i < n; i++)
         p[i] = (int)i/2;
    return p;
}

float* Array2(int n)
{
    float* p = new float[n];
    for (int i = 0; i < n; i++)
         p[i] = (float)i/2;
    return p;
}

void main()
{
    int n = 10;
    int* q;
    float* r;

    //Se genera el arreglo en la funicón Array
    q = Array(n);
    r = Array2(n);
    //Se imprime el arreglo generado
    for (int i = 0; i < n; i++)
         cout << q[i] << "\t";
    cout << endl;
    for (int i = 0; i < n; i++)
         cout << r[i] << "\t";
    _getch();
}
```

Mediante funciones plantilla es posible utilizar una sola función para distintos tipos de datos, reduciendo la duplicación de código. La sintaxis para declarar una función plantilla es:

```
template <class type>
type Nombre_De_La_Funcion
{
    //Definición de la función
    //con el código necesario
}
```

La palabra clave **type** puede sustituirse por cualquier identificador y funciona como un comodín que representa el tipo de dato sobre el cual se trabajará. Su invocación se realiza de la siguiente manera:

```
Var = Nombre_De_La_Funcion <tipo_de_dato> (argumentos);
```

El Código 3.69 ejemplifica el modo de sintetizar el Código 3.68 por medio de la implementación de funciones plantilla.

Código 3.69. Generación de dos arreglos usando una función plantilla.

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

using namespace std;

template <class type>
type *Arreglo(int n)
{
    type*p = new type[n];
    for (int i = 0; i < n; i++)
         p[i] = (type)i/2;
    return p;
}

void main()
{
    int n = 10;
    int* p;
    float* q;

    //Se crean los arreglos
    p = Arreglo <int>(n);
    q = Arreglo <float> (n);
    //Se imprimen los dos arreglos
    for (int i = 0; i < n; i++)
         cout << p[i] << "\t" << q[i] << endl;
    _getch();
}
```

Clases plantilla

Del mismo modo que las funciones pueden definirse mediante plantillas, también es posible aplicarlas a las clases. La sintaxis correspondiente es la siguiente:

```
template <class type>
class Identificador
{
   //Definición de la clase
}
```

Por ejemplo, en el Código 3.70 se genera un arreglo cuyo tipo es entero, al cual se asignan los datos correspondientes desde su constructor.

Código 3.70. Generación de un arreglo tipo entero y flotante usando clases.

```
#include <iostream>
#include <conio.h>
using namespace std;

template <class type>
class Arreglo
{
private:
   int n;
public:
   type* A;
   Arreglo(int k);
   void Imprimir();
};

template <class type>
Arreglo<type>::Arreglo(int k)
{
   n = k;
   A = new type[n];
   for (int i = 0; i < n; i++)
        A[i] = (type)i / 2;
}

template <class type>
void Arreglo<type>::Imprimir()
{
   for (int i = 0; i < n; i++)
        cout << A[i] << "\t";
   cout << endl;
}

void main()
{
   Arreglo <int> a(10);
   Arreglo <float> b(10);
   a.Imprimir();
   b.Imprimir();
   _getch();
}
```

Como alternativa al procedimiento del Código 3.70 se presenta en el Código 3.71. En dicho caso, al declarar el *template* puede enviarse un parámetro adicional, utilizado para definir el tamaño del arreglo. La operación es viable porque las plantillas admiten múltiples valores, lo que permite construir estructuras más robustas.

Código 3.71. Declaración de una plantilla que incorpora múltiples parámetros.

```
#include <iostream>
#include <conio.h>
using namespace std;

template <class type, int N>
class Arreglo
{
private:
   int n;
public:
   type* A;
   Arreglo();
   void Imprimir();
};

template <class type, int N>
Arreglo<type, N>::Arreglo()
{
   n = N;
   A = new type[n];
   for (int i = 0; i < n; i++)
        A[i] = (type)i / 2;
}

template <class type, int N>
void Arreglo<type, N>::Imprimir()
{
   for (int i = 0; i < n; i++)
        cout << A[i] << "\t";
   cout << endl;
}

void main()
{
   Arreglo <int, 10> a;
   Arreglo <float, 5> b;
   a.Imprimir();
   b.Imprimir();
   _getch();
}
```

3.2.15. FUNCIONES AMIGAS

Estas funciones se distinguen por disponer de acceso a todos los miembros de una clase, si bien son externas a ella, incluso aquellos declarados como privados (**`private`**) o protegidos (**`protected`**). Para definir una función amiga, se antepone la palabra **`friend`** en su declaración. El Código 3.72 presenta un ejemplo del uso de funciones amigas.

Código 3.72. Ejemplo del uso de funciones amigas.

```
#include <iostream>

using namespace std;

class MiClase
{
private:
    int a;
public:
    void printMembers();
    friend void funcionAmiga(int x, MiClase& mc); //Se usa la palabra friend y uno
                          //  de los parámetros es una referencia a la clase
MiClase
};

void MiClase::printMembers()
{
    cout << "El valor de 'a' es: " << a << endl;
}

void funcionAmiga(int x, MiClase& mc)
{
    mc.a = x; //Acceso a un miembro privado del objeto mc de la clase MiClase
}

int main()
{
    MiClase obj;
    funcionAmiga(5, obj);
    obj.printMembers();

    return 0;
}
```

Con este libro se busca ofrecer a los estudiantes un punto de partida sólido en la programación con lenguaje C++. Una vez que el estudiante asimile los fundamentos del lenguaje, la transición hacia otros entornos de programación orientada a objetos resultará más accesible y consonante.

REFERENCIAS

[1] Acera García, M. A. (2017). *Curso de programación* (3ª ed.). Anaya Multimedia.

[2] Aisa, C. B., Corres Sanz, J. M. y Ruiz Zamarreño, C. R. (2017). *Programación de microcontroladores PIC en lenguaje C*. Alfaomega - Marcombo.

[3] Alcolea Huertos, A. (2023). *La historia de los lenguajes de programación.* Computer Hoy. https://computerhoy.20minutos.es/reportajes/tecnologia/historia-lenguajes-programacion-428041

[4] Balagurusamy, E. (2001). *Object-Oriented Programming with C++*. McGraw-Hill Education.

[5] Bates, M. P. (2008). *Programming 8-bit PIC microcontrollers in C: With Interactive Hardware Simulation.* George Newnes Ltd. https://doi.org/10.1016/B978-0-7506-8960-1.X0001-6

[6] García Breijo, E. (2009). *Compilador C CCS y simulador PROTEUS para microcontroladores PIC.* Alfaomega - Marcombo.

[7] Ceballos Sierra, F. J. (2015). *C/C++ Curso de programación* (4ª ed.). RA-MA.

[8] Deitel, H. M. y Deitel, P. J. (2003). *Cómo programar en C++* (4ª ed.). Prentice Hall.

[9] Calvo Álvarez, P. (2019). *SOS: El planeta necesita programadores*. Cinco Días. https://cincodias.elpais.com/cincodias/2019/07/12/fortunas/1562946302_797482.html

[10] European Instiute Of Technology. (2022). *Evolución de los lenguajes de programación: Inicio y actualidad*. Epitech. https://www.epitech-it.es/evolucion-lenguajes-de-programacion/

[11] Galeano, G. (2009). *Programación de sistemas embebidos en C.* Alfaomega.

[12] Wilson Kernighan B. y McAlistair Ritchie, D. (1991). *El lenguaje de programación C*. (2ª ed.). Prentice Hall.

[13] Lores Gónzales, M. A. (2022). *Con tanta demanda de programadores ¿cómo es posible que no encuentre trabajo si sé programar?* Campus MVP.es. https://www.campusmvp.es/recursos/post/con-tanta-demanda-de-programadores-como-es-posible-que-no-encuentre-trabajo-si-se-programar.aspx

[14] Singh Malik, D. (2013). *Estructuras de datos con C++* (2ª ed.). Cengage Learning.

[15] Ramos Arreguín J. M. y Aceves Fernández M. A. (2010). *Lógica digital*. Universidad Autónoma de Querétaro.

[16] Rao, S. (2012). *Sams Teach Yourself C++ in One Hour a Day* (7ª ed.). Sams Publishing.

[17] Rockcontent. (2018). *Conoce los tipos de lenguaje de programación más usados en la actualidad*. rockcontent. https://es.scribd.com/document/448201290/25-tipos-de-lenguaje-de-programacion-mas-usados-en-la-actualidad

[18] Rojas, E. (2011). *La historia de los lenguajes de programación*. MuyComputerPRO. https://www.muycomputerpro.com/2011/08/26/historia-lenguajes-programacion
[19] Romero, E. P. (2004). *Fundamentos de programación C/C++* (4ª ed.). Alfaomega.
[20] Schildt, H. (2003). *C++: The complete reference* (4ª ed.). McGraw-Hill Education.
[21] Stroustrup, B. (2014). *A Tour of C++*. Addison-Wesley.
[22] Subero, A. (2017). *Programming PIC Microcontrollers with XC8*. Apress.
[23] Sznajdleder, P. A. (2017). *Algoritmos a fondo: Con implementaciones en C y Java.* Alfaomega.
[24] Alatorre, K. (2019). *Falta de programadores representa un problema de desarrollo económico para México*. Universidad de Guadalajara. https://www.udg.mx/es/noticia/falta-de-programadores-representa-un-problema-de-desarrollo-economico-para-mexico

Anexos

4. Anexos

4.1. SISTEMAS NUMÉRICOS BINARIO, OCTAL Y HEXADECIMAL

Seguramente el lector está familiarizado con el sistema numérico decimal, puesto que es el que utilizamos día a día. A pesar de ello, trabajar con sistemas digitales, cuyo procesamiento de la información recae en principios matemáticos, conlleva repensar el modo de manejar la numeración.

Una computadora es incapaz de llevar a cabo conteos en el sistema decimal; en su lugar, aplica un sistema rudimentario de contabilización basado en la presencia (1) o ausencia (0) de energía; es decir, implementa una representación binaria de la información. Además, es necesario considerar que las capacidades aritméticas de la computadora se limitan solamente a la adición y la multiplicación; por tal motivo, para realizar cualquier otro tipo de operaciones, los sistemas digitales recurren a la utilización de algoritmos que se basan en esos dos procedimientos básicos.

Por tanto, es imprescindible para cualquier desarrollador de software conocer los sistemas de numeración binario, octal y hexadecimal, así como su relación con el sistema decimal.

4.2. BASE NUMÉRICA BINARIA

Se trata de la base numérica más simple para un sistema digital. Se caracteriza por estar constituido por únicamente dos guarismos o símbolos numéricos: los elementos contenidos en el conjunto {0, 1}.

4.2.1. CONVERSIÓN DE DECIMAL A BINARIO PARA VALORES ENTEROS

Aunque el proceso de conversión es aplicable a cualquier número, en el contexto de la programación digital, debe determinarse la cantidad de cifras (*Nb*) necesaria para representar una cantidad (*N*) dada; primero, porque ayuda a optimizar el aprovechamiento de recursos; y segundo, porque las cantidades que exceden el límite terminan truncadas cuando los bits son insuficientes. Dicha tarea se lleva a cabo por medio de la fórmula siguiente:

$$Nb = \frac{\log(N)}{\log(2)}$$

Donde

Nb número de cifras; si es fraccionario, se toma el entero siguiente, pues no existen las fracciones de bits.

N cantidad a convertir

A continuación, se presentan ejemplos de conversión de datos decimales a binarios, tanto con bits definidos como indefinidos.

Ejemplo 1. Convertir el valor decimal 213 a binario.

1. Se divide 213 entre 2, y el residuo es 1. El nuevo valor entero es 106.
2. Se divide 106 entre 2, y el residuo es 0. El nuevo valor entero es 53.
3. Se divide 53 entre 2, y el residuo es 1. El nuevo valor entero es 26.
4. Se divide 26 entre 2, y el residuo es 0. El nuevo valor entero es 13.
5. Se divide 13 entre 2, y el residuo es 1. El nuevo valor entero es 6.
6. Se divide 6 entre 2, y el residuo es 0. El nuevo valor entero es 3.
7. Se divide 3 entre 2, y el residuo es 1. El nuevo valor entero es 1.
8. Se divide 1 entre 2, y el residuo es 1. El nuevo valor entero es 0, por lo tanto, el proceso de conversión se detiene.

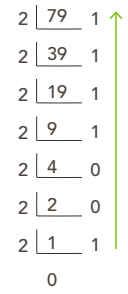
2	213	1
2	106	0
2	53	1
2	26	0
2	13	1
2	6	0
2	3	1
2	1	1
	0	

El resultado se forma al encadenar los residuos, del último al primero:

$$213_{10} \rightarrow 11010101_{2}$$

Ejemplo 2. Convertir el valor decimal 79 a binario.

1. Se divide 79 entre 2, y el residuo es 1. El nuevo valor entero es 39.
2. Se divide 39 entre 2, y el residuo es 1. El nuevo valor entero es 19.
3. Se divide 19 entre 2, y el residuo es 1. El nuevo valor entero es 9.
4. Se divide 9 entre 2, y el residuo es 1. El nuevo valor entero es 4.
5. Se divide 4 entre 2, y el residuo es 0. El nuevo valor entero es 2.
6. Se divide 2 entre 2, y el residuo es 0. El nuevo valor entero es 1.
7. Se divide 1 entre 2, y el residuo es 1. El nuevo valor entero es 0, por lo tanto, el proceso de conversión se detiene.

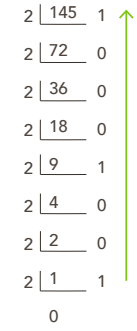


El resultado se forma al encadenar los residuos, del último al primero:

$$79_{10} \rightarrow 1001111_2$$

Ejemplo 3. Convertir el valor decimal 145 a binario con 8 bits.

1. Se divide 145 entre 2, y el residuo es 1. El nuevo valor entero es 72.
2. Se divide 72 entre 2, y el residuo es 0. El nuevo valor entero es 36.
3. Se divide 26 entre 2, y el residuo es 0. El nuevo valor entero es 18.
4. Se divide 18 entre 2, y el residuo es 0. El nuevo valor entero es 9.
5. Se divide 9 entre 2, y el residuo es 1. El nuevo valor entero es 4.
6. Se divide 4 entre 2, y el residuo es 0. El nuevo valor entero es 2.
7. Se divide 2 entre 2, y el residuo es 0. El nuevo valor entero es 1.
8. Se divide 1 entre 2, y el residuo es 1. El nuevo valor entero es 0, por lo tanto, el proceso de conversión se detiene.

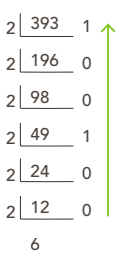


El resultado se forma tomando los residuos del último al primero, y queda:

$$145_{10} \rightarrow 10010001_2$$

Ejemplo 4. Convertir el valor decimal 393 a binario con 6 bits.

1. Se divide 393 entre 2, y el residuo es 1. El nuevo valor entero es 196.
2. Se divide 196 entre 2, y el residuo es 0. El nuevo valor entero es 98.
3. Se divide 98 entre 2, y el residuo es 0. El nuevo valor entero es 49.
4. Se divide 49 entre 2, y el residuo es 1. El nuevo valor entero es 24.
5. Se divide 24 entre 2, y el residuo es 0. El nuevo valor entero es 12.
6. Se divide 12 entre 2, y el residuo es 0. El nuevo valor entero es 6.



Puesto que se han agotado los bits designados para representar cada cifra, el proceso se trunca antes de alcanzar el resultado deseado.

$$393_{10} \rightarrow 001001_2$$

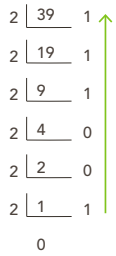
4.2.2. CONVERSIÓN DE DECIMAL A BINARIO PARA VALORES FRACCIONARIOS.

Cuando la conversión se realiza para valores expresados con punto decimal, el proceso se divide en dos etapas; una para la parte entera y otra dedicada a la fraccionaria. En este escenario suele ser imprescindible asignar una cantidad de bits para los valores decimales; de lo contrario, el proceso puede extenderse indefinidamente.

Ejemplo 5. Convertir el valor decimal 39.3 a binario con 6 bits de parte fraccionaria.

Primero, se convierte la parte entera.

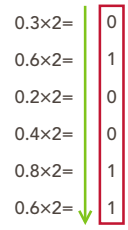
1. Se divide 39 entre 2, y el residuo es 1. El nuevo valor entero es 19.
2. Se divide 19 entre 2, y el residuo es 1. El nuevo valor entero es 9.
3. Se divide 9 entre 2, y el residuo es 1. El nuevo valor entero es 4.
4. Se divide 4 entre 2, y el residuo es 0. El nuevo valor entero es 2.
5. Se divide 2 entre 2, y el residuo es 0. El nuevo valor entero es 1.
6. Se divide 1 entre 2, y el residuo es 1. El nuevo valor entero es 0.



$$39_{10} \rightarrow 100111_2$$

Enseguida, se determina el equivalente binario de la parte fraccional. Se toma solamente la parte fraccionaria y se duplica. Si el resultado es mayor o igual a 1, se asigna un 1 al resultado, de lo contrario, se asigna 0. El proceso se detiene hasta que el resultado de la multiplicación se reduzca a cero o se agoten los bits destinados a la parte fraccionaria. La representación en sistema binario se forma al encadenar los productos de las multiplicaciones consecutivas.

1. Se multiplica 0.3 por 2, y el resultado es 0.6. Se asigna 0.
2. Se multiplica 0.6 por 2, y el resultado es 1.2. Se asigna 1.
3. Se multiplica 0.2 por 2, y el resultado es 0.4. Se asigna 0.
4. Se multiplica 0.4 por 2, y el resultado es 0.8. Se asigna 0.
5. Se multiplica 0.8 por 2, y el resultado es 1.6. Se asigna 1.
6. Se multiplica 0.6 por 2, y el resultado es 1.2. Se asigna 1.



La conversión de la parte decimal es:

$$0.3_{10} \rightarrow 0.010011_2$$

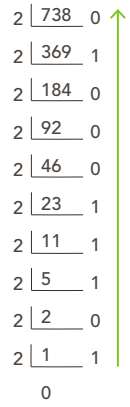
Combinando los dos resultados, se obtiene:

$$39.3_{10} \rightarrow 100111.010011_2$$

Ejemplo 6. Convertir el valor decimal 738.9457 a binario con 8 bits de parte fraccionaria.

Primero, se convierte la parte entera.

1. Se divide 738 entre 2, el residuo es 0.
2. Se divide 369 entre 2, el residuo es 1.
3. Se divide 184 entre 2, el residuo es 0.
4. Se divide 92 entre 2, el residuo es 0.
5. Se divide 46 entre 2, el residuo es 0.
6. Se divide 23 entre 2, el residuo es 1.
7. Se divide 11 entre 2, el residuo es 1.
8. Se divide 5 entre 2, el residuo es 1.
9. Se divide 2 entre 2, el residuo es 0.
10. Se divide 1 entre 2, el residuo es 1.



$$738_{10} \rightarrow 1011100010_2$$

Enseguida, se obtiene el equivalente en binario de la parte fraccional 0.9457.

1.	Se multiplica 0.9457 por 2 = 1.8914. Se asigna 1.	0.9457×2=	1	.8914
2.	Se multiplica 0.8914 por 2 = 1.7828. Se asigna 1.	0.8914×2=	1	.7828
3.	Se multiplica 0.7828 por 2 = 1.5656. Se asigna 1.	0.7828×2=	1	.5656
4.	Se multiplica 0.5656 por 2 = 1.1312. Se asigna 1.	0.5656×2=	1	.1312
5.	Se multiplica 0.1312 por 2 = 0.2624. Se asigna 0.	0.1312×2=	0	.2624
6.	Se multiplica 0.2624 por 2 = 0.5248. Se asigna 0.	0.2624×2=	0	.5248
7.	Se multiplica 0.5248 por 2 = 1.0496. Se asigna 1.	0.5248×2=	1	.0496
8.	Se multiplica 0.0496 por 2 = 0.0992. Se asigna 0.	0.0496×2=	0	.0992

$$0.9457_{10} \rightarrow 0.11110010_2$$

Combinando los dos resultados, se obtiene:

$$738.9457_{10} \rightarrow 1011100010.11110010_2$$

4.2.3. CONVERSIÓN DE BINARIO A DECIMAL

En este caso el proceso es más sencillo, debido a que cada posición de los bits tiene un valor específico denotado por 2^i, donde i representa la posición de la cifra binaria; entonces, basta con sumar el peso correspondiente a cada cifra para encontrar el valor equivalente en base 10; sin embargo, es necesario aclarar que el conteo comienza desde la posición inicial $i = 0$. Asimismo, cabe mencionar que el proceso puede ayudar a verificar que una conversión de decimal a binario se ha realizado de manera correcta.

En el sistema binario, cada dígito tendrá un "peso" en la cantidad, según la posición que ocupe. Estos pesos incrementan de derecha a izquierda, dando inicio en el valor 2^0. Luego, según se desplaza hacia la izquierda, el peso se duplicará en cada nueva posición. Los dígitos 1 y 0 indican si el peso se suma o no al valor en el sistema de base 10. Bajo el mismo principio, si el desplazamiento se realiza hacia la derecha, el peso se reducirá a la mitad; los valores detrás del punto decimal asumirán el peso 2^{-1}, 2^{-2}, 2^{-3}, y así de manera sucesiva. Los ejemplos mostrados a continuación ilustran el proceso en forma estructurada.

Ejemplo 7. Convertir el valor binario 11010101 a decimal.

CANTIDAD	1	1	0	1	0	1	0	1
ASIGNACIÓN DE PESOS	1×2^7	1×2^6	0×2^5	1×2^4	0×2^3	1×2^2	0×2^1	1×2^0
SUMA EN SISTEMA DECIMAL	128	64	0	16	0	4	0	1
RESULTADO EN SISTEMA DECIMAL	213							

Ejemplo 8. Convertir el valor binario 1001111 a decimal.

CANTIDAD	1	0	0	1	1	1	1
ASIGNACIÓN DE PESOS	1×2^6	0×2^5	0×2^4	1×2^3	1×2^2	1×2^1	1×2^0
SUMA EN SISTEMA DECIMAL	64	0	0	8	4	2	1
RESULTADO EN SISTEMA DECIMAL	79						

Ejemplo 9. Convertir el valor binario 10010001 a decimal.

CANTIDAD	1	0	0	1	0	0	0	1
ASIGNACIÓN DE PESOS	1×2^7	0×2^6	0×2^5	1×2^4	0×2^3	0×2^2	0×2^1	1×2^0
SUMA EN SISTEMA DECIMAL	128	0	0	16	0	0	0	1
RESULTADO EN SISTEMA DECIMAL	145							

Ejemplo 10. Convertir el valor binario 100111.010011 a decimal.

CANTIDAD	1	0	0	1	1	1	.	0	1	0	0	1	1
ASIGNACIÓN DE PESOS	1×2^5	0×2^4	0×2^3	1×2^2	1×2^1	1×2^0	.	0×2^{-1}	1×2^{-2}	0×2^{-3}	0×2^{-4}	1×2^{-5}	1×2^{-6}
SUMA EN SISTEMA DECIMAL	32	0	0	4	2	1	.	0	0.25	0	0	0. 03125	0. 015625
RESULTADO EN SISTEMA DECIMAL	39.286875												

4.3. BASE NUMÉRICA OCTAL

Se caracteriza por estar formada por los 8 guarismos que constituyen el conjunto {0, 1, 2, 3, 4, 5, 6, 7}, y porque, a diferencia del sistema binario, en el octal, cada posición representa una potencia de 8.

4.3.1. CONVERSIÓN DE DECIMAL A OCTAL PARA VALORES ENTEROS

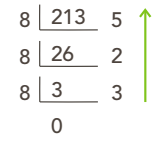
Si bien el proceso es similar al que se lleva a cabo durante la conversión del sistemas decimal al binario, existe una diferencia en tanto que, en lugar de dividir entre 2, las divisiones sucesivas se realizan entre 8. El número de cifras requeridas (*No*) para representar un valor *N* en

el sistema de base ocho puede calcularse con la fórmula siguiente; si ocurre el caso de que *No* sea fraccionario, se toma el entero mayor, por la misma razón que se ha especificado en apartados anteriores.

$$No = \frac{\log (N)}{\log (8)}$$

Ejemplo 11. Convertir el valor decimal 213 a octal.

1. Se divide 213 entre 8, el resultado es 26, con residuo 5.
2. Se divide 26 entre 8, el resultado es 3, con residuo 2.
3. Se divide 3 entre 8, y el resultado es 0, con residuo 3.

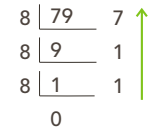


Por lo tanto:

$$213_{10} \rightarrow 325_{8}$$

Ejemplo 12. Convertir el valor decimal 79 a octal.

1. Se divide 79 entre 8, el resultado es 9, con residuo 7.
2. Se divide 9 entre 8, el resultado es 1, con residuo 1.
3. Se divide 1 entre 8, el resultado es 0, con residuo 1.

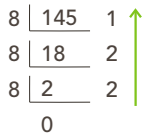


Por lo tanto:

$$79_{10} \rightarrow 117_{8}$$

Ejemplo 13. Convertir el valor decimal 145 a octal con 8 bits.

1. Se divide 145 entre 8, el resultado es 18, con residuo 1.
2. Se divide 18 entre 8, el resultado es 2, con residuo 2.
3. Se divide 2 entre 8, el resultado es 0, con residuo 2.



Por lo tanto:

$$145_{10} \rightarrow 221_{8}$$

4.3.2. CONVERSIÓN DE DECIMAL A OCTAL PARA VALORES FRACCIONARIOS

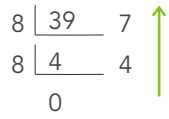
Se trata de un proceso dividido en dos tapas, similar al que conforma la conversión de base diez a base dos: primero la parte entera y posteriormente la parte fraccionaria. En los ejemplos siguientes se muestra este tipo de conversiones.

Ejemplo 14. Convertir el valor decimal 39.3 a octal con 4 símbolos de parte fraccionaria.

Conversión de la parte entera.

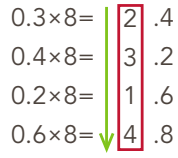
1. Se divide 39 entre 8, el resultado es 4, con residuo 7.
2. Se divide 4 entre 8, el resultado es 0, con residuo 4.

$$39_{10} \rightarrow 47_{8}$$



Conversión de la parte fraccionaria.

1. Se multiplica 0.3 por 8, resultado 2.4, se toma el 2.
2. Se multiplica 0.4 por 8, resultado 3.2, se toma el 3.
3. Se multiplica 0.2 por 8, resultado 1.6, se toma el 1.
4. Se multiplica 0.6 por 8, resultado 4.8, se toma el 4.



$$0.3_{10} \rightarrow 0.2314_{8}$$

Combinando los dos resultados, se obtiene:

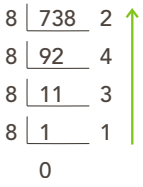
$$39.3_{10} \rightarrow 47.2314_{8}$$

Ejemplo 15. Convertir el valor decimal 738.9457 a octal con 6 símbolos de parte fraccionaria.

Conversión de la parte entera.

1. Se divide 738 entre 8, el resultado es 92, con residuo 2.
2. Se divide 92 entre 8, el resultado es 11, con residuo 4.
3. Se divide 11 entre 8, el resultado es 1, con residuo 3.
4. Se divide 1 entre 8, el resultado es 0, con residuo 1.

$$738_{10} \rightarrow 1342_{8}$$



Conversión de la parte fraccionaria.

1. Se multiplica 0.9457 por 8, resultado 7.5656, se toma el 7.
2. Se multiplica 0.5656 por 8, resultado 4.5248, se toma el 4.
3. Se multiplica 0.5248 por 8, resultado 4.1984, se toma el 4.
4. Se multiplica 0.1984 por 8, resultado 1.5872, se toma el 1.
5. Se multiplica 0.5872 por 8, resultado 4.6976, se toma el 4.
6. Se multiplica 0.6976 por 8, resultado 5.5808, se toma el 5.

$0.9457 \times 8 = 7.5656$
$0.5656 \times 8 = 4.5248$
$0.5248 \times 8 = 4.1984$
$0.1984 \times 8 = 1.5872$
$0.5872 \times 8 = 4.6976$
$0.6976 \times 8 = 5.5808$

$$0.9457_{10} \rightarrow 0.744145_{8}$$

4.3.3. CONVERSIÓN DE OCTAL A DECIMAL

La conversión de números octales a decimales sigue el mismo principio que en el sistema binario, pero utilizando la base 8 en lugar de la base 2. Cada dígito del número octal dispone de un "peso" determinado por su posición, el cual se expresa como 8^i, donde i corresponde al índice de la posición, comenzando desde $i = 0$ en la cifra del extremo derecho. Para obtener el valor equivalente en decimal, basta con multiplicar cada dígito por su respectivo peso y sumar los resultados. Este procedimiento es útil para verificar la exactitud de una conversión realizada en sentido contrario (de decimal a octal). En síntesis, en un valor en octal, cada símbolo tiene un peso en la cantidad, como se puede observar en las tablas a continuación. El resultado decimal surge de sumar dichos pesos.

Ejemplo 16. Convertir el valor octal 325 a decimal.

CANTIDAD	3	2	5
ASIGNACIÓN DE PESOS	3×8^2	2×8^1	5×8^0
SUMA EN SISTEMA DECIMAL	192	16	5
RESULTADO EN SISTEMA DECIMAL	213		

Ejemplo 17. Convertir el valor octal 117 a decimal.

CANTIDAD	1	1	7
ASIGNACIÓN DE PESOS	1×8^2	1×8^1	7×8^0
SUMA EN SISTEMA DECIMAL	64	8	7
RESULTADOEN SISTEMA DECIMAL	79		

Ejemplo 18. Convertir el valor octal 221 a decimal.

CANTIDAD	2	2	1
ASIGNACIÓN DE PESOS	2×8^2	2×8^1	1×8^0
SUMA EN SISTEMA DECIMAL	128	16	1
RESULTADO EN SISTEMA DECIMAL	145		

Ejemplo 19. Convertir el valor octal 47.23 a decimal.

CANTIDAD	4	7	.	2	3
ASIGNACIÓN DE PESOS	4×8^1	7×8^0	.	2×8^{-1}	3×8^{-2}
SUMA EN SISTEMA DECIMAL	32	7	.	0.25	0.046875
RESULTADO EN SISTEMA DECIMAL	39.296875				

4.4. BASE NUMÉRICA HEXADECIMAL

Se caracteriza por estar formada por 16 símbolos numéricos, cada uno de los cuales ocupa una posición que representa una potencia de base 16. Estos símbolos son los elementos del conjunto {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} (las letras pueden ser usadas indistintamente como minúsculas o mayúsculas). Al tiempo de realizar la conversión, si el resultado

del residuo es mayor que 9, entonces se coloca el equivalente en hexadecimal del residuo. Como ejemplo, si el residuo es 13, se le asigna la letra D. La correspondencia de las letras con la base decimal se muestra en la siguiente tabla.

DECIMAL	HEXADECIMAL
10	A
11	B
12	C
13	D
14	E
15	F

4.4.1. CONVERSIÓN DE DECIMAL A HEXADECIMAL PARA VALORES ENTEROS

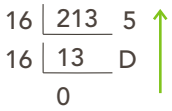
El proceso es similar a la conversión a binario, a reserva de que se la serie de divisiones se realiza entre 16. El número de símbolos que se requieren para representar un valor *N* en hexadecimal se puede calcular con la fórmula mostrada a continuación. Tal como ocurre en los métodos anteriores, si *Nh* es fraccionario, se toma el entero mayor.

$$Nh = \frac{\log(N)}{\log(16)}$$

Por ejemplo, para representar la cifra $N = 5341_{10}$ en hexadecimal, entonces se obtiene $Nh = 3.095$. Se redondea al inmediato superior, por lo tanto, $Nh = 4$.

Ejemplo 20. Convertir el valor decimal 213 a hexadecimal.

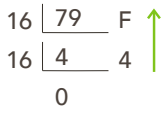
1. Se divide 213 entre 16, el resultado es 13, con residuo 5.
2. Se divide 13 entre 16, el resultado es 0, con residuo 13 (D).



$$213_{10} \rightarrow D5_{16}$$

Ejemplo 21. Convertir el valor decimal 79 a hexadecimal.

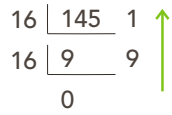
1. Se divide 79 entre 16, el resultado es 4, con residuo 15 (F).
2. Se divide 4 entre 16, el resultado es 0, con residuo 4.



$$79_{10} \rightarrow 4F_{16}$$

Ejemplo 22. Convertir el valor decimal 145 a hexadecimal con 8 bits.

1. Se divide 145 entre 16, el resultado es 9, con residuo 1.
2. Se divide 9 entre 16, el resultado es 0, con residuo 9.



$$145_{10} \rightarrow 91_{16}$$

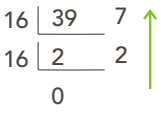
4.4.2. CONVERSIÓN DE DECIMAL A OCTAL PARA VALORES FRACCIONARIOS

Del mismo modo que en las técnicas de conversión anteriores, cuando se trabaja con números fraccionarios, se resuelve primero la parte entera y posteriormente la fraccionaria. En los siguientes ejemplos se muestra este tipo de conversiones.

Ejemplo 23. Convertir el valor decimal 39.3 a hexadecimal con 4 símbolos de parte fraccionaria.

Conversión de la parte entera.

1. Se divide 39 entre 16, el resultado es 2, con residuo 7.
2. Se divide 2 entre 16, el resultado es 0, con residuo 2.



$$39_{10} \rightarrow 27_{16}$$

Conversión de la parte fraccionaria.

1. Se multiplica 0.3 por 16, resultado 4.8, se toma el 4.
2. Se multiplica 0.8 por 16, resultado 12.8, se toma el 12 = C.
3. Se multiplica 0.8 por 16, resultado 12.8, se toma el 12 = C.
4. Se multiplica 0.8 por 16, resultado 12.8, se toma el 12 = C.

0.3×16 = 4.8 = 4 .8
0.8×16 = 12.8 = C .8
0.8×16 = 12.8 = C .8
0.8×16 = 12.8 = C .8

$$0.3_{10} \rightarrow 0.4CCC_{16}$$

Combinando los dos resultados, se obtiene:

$$39.3_{10} \rightarrow 27.4CCC_{16}$$

Ejemplo 24. Convertir el valor decimal 738.9457 a hexadecimal con 6 símbolos de parte fraccionaria.

Primero, se convierte la parte entera.

1. Se divide 738 entre 16, el resultado es 46, con residuo 2.
2. Se divide 46 entre 16, el resultado es 2, con residuo 14 = E.
3. Se divide 2 entre 16, el resultado es 0, con residuo 2.

16 | 738 2
16 | 46 E
16 | 2 2
0

$$738_{10} \rightarrow 2E2_{16}$$

Segundo, conversión de la parte fraccionaria.

1. 0.9457 por 16 = 15.1312, se toma el 15 = F.
2. 0.1312 por 16 = 2.0992, se toma el 2.
3. 0.0992 por 16 = 1.5872, se toma el 1.
4. 0.5872 por 16 = 9.3952, se toma el 9.
5. 0.3952 por 16 = 6.3232, se toma el 6.
6. 0.3232 por 16 = 5.1712, se toma el 5.

0.9457×16 = 15.1312 = F .1312
0.1312 ×16 = 2.0992 = 2 .0992
0.0992 ×16 = 1.5872 = 1 .5872
0.5872×16 = 9.3952 = 9 .3952
0.3952×16 = 6.3232 = 6 .3232
0.3232×16 = 5.1712 = 5 .1712

$$0.9457_{10} \rightarrow 0.F21965_{16}$$

Combinando los dos resultados, se obtiene:

$$738.9457_{10} \rightarrow 2E2.F21965_{16}$$

4.4.3. CONVERSIÓN DE HEXADECIMAL A DECIMAL

Es un proceso de asignación de pesos similar a los ya descritos; como es de esperarse, en este caso, cada posición representa una potencia de 16.

Ejemplo 25. Convertir el valor hexadecimal D5 a decimal.

CANTIDAD	D	5
ASIGNACIÓN DE PESOS	13×16^1	5×16^0
SUMA EN SISTEMA DECIMAL	208	5
RESULTADO EN SISTEMA DECIMAL	213	

Ejemplo 26. Convertir el valor hexadecimal 4F a decimal.

CANTIDAD	4	F
ASIGNACIÓN DE PESOS	4×16^1	15×16^0
SUMA EN SISTEMA DECIMAL	64	15
RESULTADO EN SISTEMA DECIMAL	79	

Ejemplo 27. Convertir el valor hexadecimal 91 a decimal.

CANTIDAD	9	1
ASIGNACIÓN DE PESOS	9×16^1	1×16^0
SUMA EN SISTEMA DECIMAL	144	1
RESULTADO EN SISTEMA DECIMAL	145	

Ejemplo 28. Convertir el valor hexadecimal 27.4CCC a decimal.

CANTIDAD	2	7	.	4	C	C	C
ASIGNACIÓN DE PESOS	2×16^1	7×16^0	.	4×16^{-1}	12×16^{-2}	12×16^{-3}	12×16^{-4}
RESULTADO EN SISTEMA DECIMAL	39.299987						

La presente edición de
Programación digital
fue maquetada por Mariana Cano León
en el Despacho de Publicaciones de la Facultad de Ingeniería
de la Universidad Autónoma de Querétaro.
El cuidado de la edición estuvo a cargo de Soid Ruiz Ramírez.
Se publicó en Santiago de Querétaro, Qro.,
el 3 de julio de 2026.

Universidad Autónoma de Querétaro
Facultad de Ingeniería

LIBROS DE TEXTO
FACULTAD DE INGENIERÍA

Programación digital se presenta como una guía práctica para aprender a programar en el lenguaje C++, abarcando los conceptos fundamentales. Este volumen ofrece fundamentos sólidos y claros, así como explicaciones pormenorizadas de los principios básicos de la programación en C++, ideales para aquellos que se inician en este campo. Se incluye una amplia gama de ejemplos prácticos y ejercicios para fortalecer la comprensión y facilitar la aplicación de los conceptos en situaciones reales.

Consta de una introducción detallada a temas avanzados, como son la programación orientada a objetos, gestión de memoria y estructuras de datos complejas. Las actividades propuestas permiten a los lectores poner a prueba sus habilidades y desarrollar un portafolio de programación propio.

Programación digital es una herramienta esencial para estudiantes y entusiastas de la programación que buscan adquirir o perfeccionar sus habilidades en C++. Con un enfoque didáctico, este libro guía al lector paso a paso en su formación para convertirse en un programador competente y seguro en C++.