

Tema 4. Estructuras de Control

1. Programación Estructurada

La programación estructurada es una forma de escribir programas de forma clara, utilizando únicamente tres estructuras: **secuencia**, **selección** e **iteración** o **repetitivas**; siendo innecesario y no permitiéndose el uso de instrucciones de transferencia incondicional (Por ejemplo, GoTo).

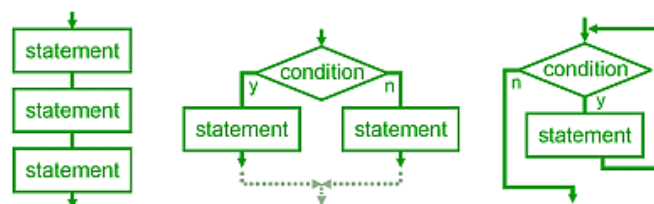
La programación estructurada surge a finales de los años 60 con el objetivo de realizar programas confiables y eficientes, y comprensibles.

En la actualidad, las aplicaciones informáticas son mucho más ambiciosas que las necesidades de aquellos años, por lo que se desarrollaron nuevas metodologías, tales como la programación orientada a objetos, y el desarrollo de entornos de programación que facilitan la programación.

De todas formas, el paradigma estructurado constituye una buena forma de iniciarse en la programación de computadoras, por lo que en este tema se verán las características de las estructuras que lo componen.

El teorema del programa estructurado de Böhm-Jacopini, demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

- **Secuencia**
- **Selección** (Instrucción condicional)
- **Iteración** (bucle de instrucciones con condición)



Solamente con estas tres estructuras o “patrones lógicos” se pueden escribir programas. En general, los lenguajes de programación tienen otras estructuras de control, basadas en o variantes de estas tres estructuras básicas.

2. Estructuras de Control

El flujo de control de un programa

La expresión **flujo de control** hace referencia al orden en el que se ejecutarán las instrucciones de un programa, desde el inicio hasta que finaliza. El flujo normal de ejecución es el **secuencial**. Si no se especifica lo contrario, la ejecución de un programa comenzaría por la primera instrucción e iría procesando una a una en el orden en que aparecen, hasta llegar a la última.

Algunos programas muy simples pueden escribirse sólo con este flujo unidireccional. En general, el programador necesitará que el programa no se comporte sólo de modo secuencial. Por ejemplo, si el programa tiene que calcular una bonificación sólo para los empleados con más de 10 años de antigüedad.

También puede ocurrir que interese que un grupo de instrucciones se ejecute repetidamente hasta que se le indique que se detenga. Por ejemplo, calcular el promedio de notas para cada uno de los alumnos de un curso.

Para las dos situaciones planteadas existen dos soluciones: las sentencias de control **selectivas** y las **iterativas** o **repetitivas**. Éstas permiten que el flujo secuencial del programa sea modificado. También cumplen con este objetivo las sentencias denominadas de **invocación** o **salto**.

Las sentencias **selectivas** o **alternativas** se denominan así porque permiten **seleccionar** uno de entre varios caminos por donde seguirá la ejecución del programa. En algunos casos esta selección viene determinada por la evaluación de una expresión lógica.

Este tipo de sentencias se clasifican en:

- simples: **SI** (IF)
- dobles: **SI-SINO** (IF-ELSE)
- múltiples: **SEGÚN-SEA** (SWITCH, CASE)

A las sentencias **iterativas** se las conoce también como sentencias **repetitivas** dado que permiten realizar algo varias veces (**repetir**, **iterar**). Dentro de ellas se distinguen:

- **PARA** (FOR)
- **MIENTRAS** (WHILE)
- **HACER-MIENTRAS** (DO WHILE))

Las sentencias de **salto** o **invocación** permiten realizar saltos en el flujo de control de un programa, es decir, permiten transferir el control del programa, alterando bruscamente el flujo de control del mismo. En programación estructurada se considera una mala práctica el uso de las condiciones de salto, ya que, entre otras cosas, restan legibilidad al código. Sin embargo, si bien se debe evitar su uso, la mayoría de los lenguajes las incluyen.

Algunas sentencias de **salto** o **invocación** son:

- **ROMPER** (BREAK)
- **CONTINUAR** (CONTINUE)
- **IR-A** (GO TO) (Su utilización no es considerada una buena práctica en programación).
- **VOLVER** (RETURN)

Aclaración: En este material los códigos que ejemplifican la aplicación de las distintas técnicas no siguen los pasos del “método de resolución de problemas” que se les exigirá en la práctica porque pretenden enfatizar solamente la técnica. En general, el programa aparece completo para que lo puedan ejecutar y hacer modificaciones para comprobar su funcionamiento.

2.1. Estructura secuencial

La estructura secuencial es aquella en que las acciones (instrucciones) se ejecutan sucesivamente, una a continuación de otra, sin posibilidad de omitir ninguna y sin bifurcaciones. Es decir que la acción **2** no se inicia hasta haber terminado la acción **1**.

Acción 1 → Acción 2 → Acción 3 → ...

Seudocódigo de una estructura secuencial:

INICIO

< acción 1 >

< acción 2 >

< acción 3 >

FIN

Ejemplo:

Dado el radio de una circunferencia, se desea obtener su longitud y el área del círculo que determina.

ALGORITMO LongitudArea

VAR

REAL: area

REAL: longitud

INICIO

LEER radio

area= 3,141592 * radio **2

longitud = 2 * (3,141592) * radio

ESCRIBIR radio, area, longitud

FIN

2.2. Estructuras selectivas

Las **estructuras selectivas** posibilitan, como resultado de la evaluación de una condición, seleccionar la o las siguientes instrucciones a ejecutar, de entre varias posibilidades o alternativas. Reciben también el nombre de estructuras **alternativas** o **de decisión**.

Las estructuras selectivas o alternativas pueden ser:

- simples: **SI** (IF)
- dobles: **SI-SINO** (IF-ELSE)
- múltiples: **SEGÚN-SEA** (SWITCH, CASE)

La base de este tipo de estructuras es la *condición*. Para expresar una condición se utilizan las *expresiones condicionales, lógicas o booleanas*, que devuelven como resultado uno de estos valores: VERDADERO (TRUE) o FALSO (FALSE).

2.2.1. Alternativa Simple

La estructura alternativa simple de la forma **SI-ENTONCES** (**IF-THEN**) ejecuta una acción o conjunto de acciones cuando se cumple una determinada condición. Como resultado de evaluar la condición, pueden ocurrir dos cosas:

- que la condición sea *verdadera*: en cuyo caso se ejecuta la acción preestablecida
- que la condición sea *falsa*: en esta situación, el programa continúa con la siguiente instrucción.

Seudocódigo de una estructura selectiva simple (en español y en inglés) y formato en C:

Seudo	Formato C
SI <expresión_lógica> ENTONCES < acción 1 > < acción 2 > < ... > < acción n > FIN-SI	if (expresión_lógica) { instrucciones; }

Nótese que las líneas correspondientes a las acciones se encuentran *indentadas*¹ o *sangradas* respecto a las palabras reservadas **SI** y **FIN-SI**, facilitando la legibilidad del código.

Ejemplo: Dado el radio de una circunferencia, se desea obtener su longitud y el área del círculo que determina, **siempre que el radio sea mayor a 3**.

Seudo	Código C
ALGORITMO LongitudArea VAR REAL: area REAL: longitud INICIO LEER radio SI radio > 3 ENTONCES area = 3.141592 * radio * radio longitud = 2 * (3.141592) * radio ESCRIBIR radio, area, longitud FIN-SI FIN	<pre># include <stdio.h> int main () { const float pi=3.1416; float radio, area, longitud; scanf ("%f", &radio); if (radio > 3.0) { area = radio * radio * pi; longitud = 2 * pi * radio; printf("Area %.2f \n", area); printf("Long %.2f \n", longitud); } return 0; }</pre>

Observe que los cálculos correspondientes (instrucciones), sólo se ejecutan si se cumple la condición, representada por la expresión lógica que incluye un operador relacional (>).

2.2.2. Alternativa Doble

La estructura condicional doble permite elegir entre dos opciones o alternativas posibles en función del cumplimiento o no de una determinada condición. Se representa de la siguiente forma:

Seudo	Formato C
SI <expresión_lógica> ENTONCES <bloque_de_instrucciones_1 > SINO <bloque_de_instrucciones_2 > FIN-SI	<pre>If (expresión_lógica){ instrucciones_1; } else { instrucciones_2; }</pre>

La **expresión_lógica** es la condición que se evalúa. Si la condición es **verdadera** se ejecuta el **bloque_de_instrucciones_1**, caso contrario, si la condición es **falsa**, se ejecutará el **bloque_de_instrucciones_2**.

En resumen, una **instrucción alternativa doble** (o simplemente **alternativa doble**) permite seleccionar, por medio de una condición, el siguiente bloque de instrucciones a ejecutar, de entre dos posibles.

¹ Este término significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores, para así separarlo del margen izquierdo y distinguirlo mejor en el texto; en el ámbito de la imprenta, este concepto se ha denominado *sangrado* o *sangría*.

Nótese que, como en el caso de la estructura alternativa simple, los bloques de instrucciones se encuentran indentados respecto de las palabras reservadas, dando mayor claridad al código.

Ejemplo: Dado el radio de una circunferencia, se desea obtener su longitud y el área del círculo que determina, siempre que el radio sea mayor a 3. Caso contrario, muestre en pantalla el mensaje “Fuera de rango”.

Seudo	Código C
ALGORITMO LongitudArea VAR REAL: area, longitud INICIO LEER radio SI radio > 3 ENTONCES area = 3.141592 * radio * radio longitud = 2 * (3.141592) * radio ESCRIBIR radio, area, longitud SINO ESCRIBIR “Fuera de rango” FIN-SI FIN	<pre># include <stdio.h> int main () { const float pi=3.1416; float radio, area, longitud; scanf("%f",&radio); if (radio > 3.0) { area = radio * radio * pi; longitud = 2 * pi * radio; printf ("Area %.2f \n", area); printf ("Longitud %.2f \n", longitud); } else { printf ("Fuera de rango %.2f \n", radio); } return 0; }</pre>

2.2.3. Alternativa Múltiple

Al diseñar un algoritmo puede ocurrir que se necesiten más de dos alternativas. Si bien esta situación puede resolverse con estructuras alternativas anidadas, como se verá más adelante, resta legibilidad al código cuando las alternativas son numerosas, incrementando también el riesgo de errores en la codificación.

La estructura **alternativa múltiple** es una toma de decisión especializada que permite evaluar una expresión con *n* posibles resultados, y en base al resultado seleccionar el siguiente bloque de instrucciones a ejecutar, de entre varios posibles.

Las palabras reservadas (**SWITCH**, **CASE**, etc.) utilizadas para esta estructura varían según el lenguaje de programación. La representación es la siguiente:

Seudocódigo:	Formato C:
SEGUN-SEA <expresión> <caso-1>: <bloque_de_instrucciones_1> <caso-2>: <bloque_de_instrucciones_2> ... <caso-n>: <bloque_de_instrucciones_n> SINO: <bloque_de_instrucciones_n+1> FIN-SEGUN-SEA	<pre>switch (selector) { case etiqueta1:instrucc1; case etiqueta2:instrucc2; default: instrucc-n+1; }</pre>

Dependiendo del valor obtenido al evaluar la expresión, se ejecutará un bloque de instrucciones u otro. En las listas de valores se deben escribir los valores que determinan el bloque de instrucciones a ejecutar, teniendo en cuenta que un valor sólo puede aparecer en una lista de valores.

Opcionalmente, se puede escribir un *<bloque_de_instrucciones_n+1>* después de **SINO**. Este bloque de instrucciones se ejecutará en el caso de que el valor obtenido al evaluar la expresión no se encuentre en ninguna de las listas de valores especificadas.

Funciona de la siguiente manera: La expresión a evaluar puede ser una variable o una función de varias variables. El valor de la expresión podrá ser de identidad (p. ej. Caso, = 8) o de intervalo (p. ej. Caso, < 100; también podría ser Caso, > 0 y < 100).

El caso SINO determina las instrucciones a ejecutar cuando ninguno de los casos previstos se ha cumplido. Si no procede, simplemente se deja de escribir.

Ejemplo: Diseñar un algoritmo que pida por teclado el número (dato entero) de un día de la semana, y luego muestre por pantalla el nombre (dato cadena) correspondiente a dicho día.

Nota: Si el número de día introducido es menor que 1 ó mayor que 7, se mostrará el mensaje: "ERROR: Día incorrecto".

<p>ALGORITMO DiaDeLaSemana</p> <p>VAR</p> <p> ENTERO: día</p> <p>INICIO</p> <p> ESCRIBIR "Ingrese día de la semana: "</p> <p> LEER día</p> <p> SEGUN_SEA dia</p> <p> 1: ESCRIBIR "Lunes"</p> <p> 2: ESCRIBIR "Martes"</p> <p> 3: ESCRIBIR "Miércoles"</p> <p> 4: ESCRIBIR "Jueves"</p> <p> 5: ESCRIBIR "Viernes"</p> <p> 6: ESCRIBIR "Sábado"</p> <p> 7: ESCRIBIR "Domingo"</p> <p> SINO: ESCRIBIR "ERROR: Día incorrecto"</p> <p> FIN-SEGUN-SEA</p> <p>FIN</p>	<pre>include <stdio.h> int main() { int dia; printf("Ingrese dia de la semana:"); scanf("%d",&dia); switch (dia) { case 1: printf("\nLunes"); break; case 2: printf("\nMartes"); break; case 3: printf("\nMiercoles"); break; case 4: printf("\nJueves"); break; case 5: printf("\nViernes"); break; case 6: printf("\nSabado"); break; case 7: printf("\nDomingo"); break; default: printf("ERROR: Dia incorrecto"); } }</pre>
--	---

En el código C, las sentencias *break*; hacen que una vez que se cumple una opción dentro del **switch**, se salga del mismo y se continúe la ejecución en la siguiente instrucción después del switch.

El uso de *break* es opcional, pero es habitual incluirlo después de cada evaluación. En caso de no hacerlo, se ejecutarán todas las instrucciones dentro de cualquier case (incluso aunque no se verifique) después del que ha verificado la condición, hasta encontrar una sentencia *break*; o terminar la ejecución del switch. Esto puede generar resultados contradictorios o no esperados, de ahí que en general siempre se incluyan sentencias *break*.

En C la evaluación de un intervalo de valores se realiza como se muestra en el ejemplo a continuación:

```

void comoUsarSwitch() {
    switch(nro) {
        case 1 ... 5:    printf("\nValor entre 1 y 5\n");
                        break;
        case 6 ... 10:   printf("\nValor entre 6 y 10\n");
                        break;
        default:         printf("\nValor ingresado es > a 10\n");
    }
}

```

2.3. Estructuras de decisión anidadas

Según lo expresado, las estructuras de decisión simple y doble permiten seleccionar entre dos alternativas posibles. Sin embargo, la instrucción SI-ENTONCES puede ser utilizada también en casos de selección de más de dos alternativas. Esto es posible *anidando* estas instrucciones. Es decir, una estructura SI-ENTONCES puede contener a otra, y esta a su vez a otra. La representación en pseudocódigo es la siguiente:

```

SI <condición_1> ENTONCES
    < sentencias_1 >
SINO
    SI <condición_2> ENTONCES
        < sentencias_2 >
    SINO
        SI <condición_3> ENTONCES
            < sentencias_3 >
        SINO
            ...
        FIN-SI
    FIN-SI
FIN-SI

```

Como se puede observar, el anidamiento de instrucciones alternativas permite ir descartando valores hasta llegar al bloque de instrucciones que se debe ejecutar.

En las instrucciones **SI** anidadas, las instrucciones **SINO** y **FIN-SI** se aplican automáticamente a la instrucción **SI** anterior más próxima.

A fin de que las estructuras anidadas sean más fáciles de leer, es práctica habitual aplicar sangría (indentación) al cuerpo de cada una.

Ejemplo: Un sensor toma (lee) la temperatura y de acuerdo con el rango en que se encuentre, debe emitir un mensaje. La escala es la siguiente:

Mayor que 100 → “Temperatura muy alta – Mal funcionamiento”;

Entre 50 y 100 → “Rango normal”;

Menor que 50 → “Muy frío – Apague el equipo”

ALGORITMO Sensor**VAR****ENTERO:** temperatura**INICIO****LEER** temperatura**SI** temperatura > 100 **ENTONCES****ESCRIBIR** "Temperatura muy alta – Mal funcionamiento"**SINO****SI** temperatura >= 50 **ENTONCES****ESCRIBIR** "Rango normal"**SINO****ESCRIBIR** "Muy frío – Apague equipo"**FIN-SI****FIN-SI****FIN****Ejemplo en código C:**

```

1  #include <stdio.h>
2  int main() {
3      int temperatura;
4      printf("Ingrese valor de temperatura:");
5      scanf("%d",&temperatura);
6
7      if (temperatura > 100) {
8          printf("\nTemperatura muy alta - Mal funcionamiento");
9      }
10     else {
11         if (temperatura <= 50) {
12             printf("\nRango normal");
13         }
14         else {
15             printf("\nMuy frío - Apague equipo");
16         }
17     }
18 }

```

2.4. Estructuras de control repetitivas

En muchas ocasiones la forma más apropiada de expresar un algoritmo consiste en la repetición de una misma instrucción de manera controlada, una cantidad finita de veces determinada de antemano (al diseñar el programa) o en tiempo de ejecución (cada vez que se corre el programa).

Por ejemplo: Un algoritmo, similar al de los cajeros automáticos, que solicite una clave al usuario y bloquee el acceso en caso de no ingresar la contraseña correcta luego de tres intentos. O bien, si se desea procesar datos ingresados por teclado o leídos desde un archivo, hasta que no se encuentren más datos.

Las estructuras algorítmicas que permiten realizar operaciones de este tipo se conocen con el nombre de *estructuras repetitivas* o *iterativas*.

2.4.1. El concepto de bucle

En programación se denomina **bucle** a la ejecución repetidas veces de un mismo conjunto de sentencias. Normalmente, en cada nueva ejecución varía algún elemento. Para comprender mejor el concepto de bucle imaginemos un tren que parte de la estación "inicio" y viaja hasta la estación "final"; cuando el maquinista llega a la estación "final" vuelve a la estación "inicio" y así sucesivamente. Cada vuelta que da el maquinista es una ejecución o una **iteración**, en la cual se ejecutan todas las instrucciones que hay en el trayecto comprendido entre la estación "inicio" y la estación "final". La figura 1 muestra este concepto.

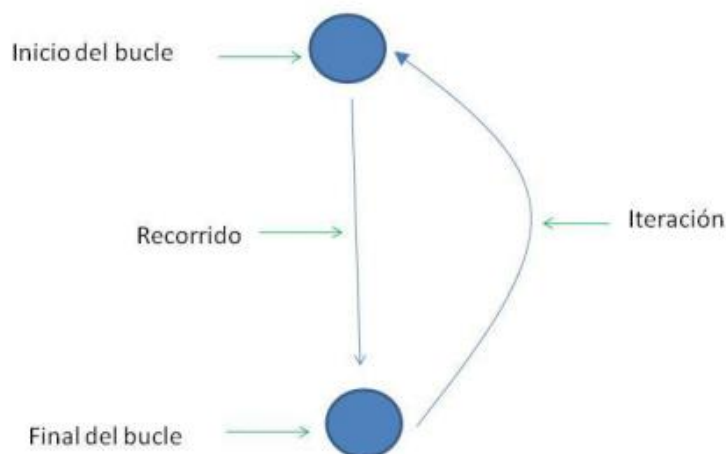


Fig. 1. Concepto de bucle. Tomado del material de Ángel Fidalgo Blanco Universidad Politécnica de Madrid <https://innovacioneducativa.files.wordpress.com/2012/10/bucles.pdf>

Para realizar un bucle correctamente, el maquinista tiene que conocer tres cosas: el comienzo, el final del bucle y el número de iteraciones que tiene que realizar. Esto es importante, si no le indicamos el número de vueltas que tiene que realizar, estaría dando vueltas indefinidamente (loop infinito).

La forma de indicar al "maquinista" el número de vueltas define el tipo de bucle.

Siempre que utilicemos las instrucciones repetitivas, tenemos que indicar al programa donde empieza el bucle, donde termina y cuántas iteraciones tiene que hacer entre el comienzo y el final. Todas las sentencias comprendidas entre el comienzo y el final del bucle se ejecutarán en cada iteración.

Para cada tipo de bucle, el comienzo, final y cantidad de iteraciones se especifica de una forma distinta. Esto también es aplicable a los distintos lenguajes de programación, cada uno define las estructuras repetitivas de forma diferente, pero el concepto es el mismo.

2.4.2. Tipos de bucles

En general se utilizan en programación, dos tipos de bucles, aquellos en los que conocemos previamente la cantidad de iteraciones, denominados, **bucles controlados por contador** y aquellos en los que no se conoce, a priori, el número exacto de iteraciones, denominados **bucles controlados por condición**.

Controlados por contador (Repetición simple)	for	Bucle de conteo cuando el número de repeticiones se conoce por anticipado y puede ser controlado por un contador; el test de la condición precede a la ejecución del cuerpo del bucle.
---	------------	--

Controlados por condición (Repetición condicional)	while	El test de condición precede a cada repetición del bucle; el cuerpo del bucle puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es falsa.
	do-while	Es adecuado cuando se debe asegurar que al menos se ejecuta el bucle una vez; el test de condición se evalúa al final.

A continuación, se describen las distintas estructuras repetitivas

2.4.3. La estructura PARA (FOR):

Se utiliza para resolver problemas en los cuales se conoce de antemano la cantidad de veces que es necesario repetir las instrucciones que componen el bucle. El formato general de la estructura en pseudo es:

PARA variable DESDE valor-inicial HASTA valor-final [INCREMENTO incremento] INSTRUCCIÓN [...] INSTRUCCIÓN FIN-PARA

Donde:

<i>variable</i>	Nombre de la variable de control de tipo numérico, en particular entero, cuyos valores se irán modificando en cada repetición.
<i>valor-inicial</i>	Valor que toma la variable en la primera repetición.
<i>valor-final</i>	Valor que toma la variable en la última repetición.
<i>incremento</i>	Incremento que recibirá la variable entre repeticiones, es decir, el valor que se le sumará a <i>variable</i> cada vez que se termine una repetición y antes de iniciar la siguiente. Si se omite, se considera que vale 1. También puede tomar valores negativos.

Cuando se ejecuta la instrucción FOR, a la variable de control se le asigna el valor inicial, al llegar al final del bucle se verifica si el valor de la variable de control es igual que el valor final; en caso negativo se incrementa el valor de la variable de control en uno (o lo que indique el incremento), y se vuelven a ejecutar todas las instrucciones del bucle, hasta que la variable de control sea igual que el valor final.

Reglas de funcionamiento del FOR:

1. Las variables de control, valor inicial y valor final deben ser todas del mismo tipo, el tipo real no está permitido (en la mayoría de los casos). Los valores iniciales y finales pueden ser tanto expresiones como constantes.
2. Antes de la primera ejecución del bucle, se le asigna el valor inicial.
3. La última ejecución del bucle ocurre cuando la variable de control es igual al valor final.
4. El valor del incremento puede ser positivo o negativo.
5. No es recomendable modificar los parámetros del FOR (variable de control, y los valores inicial y final) dentro del bucle dado que pueden generar resultados imprevistos.

Una variable de control debe ser de tipo ordinal o un subrango de tipo ordinal. Puede ser entera, tipo char o tipo lógica.

Ejemplo en pseudocódigo: Mostrar un saludo 10 veces:

```
ALGORITMO saludo
VAR
    ENTERO i
INICIO
    PARA i DESDE 1 HASTA 10 INCREMENTO +1
        ESCRIBIR "HOLA"
    FIN-PARA;
END.
```

En C, la estructura repetitiva **for** difiere de otros lenguajes de programación en el uso de los parámetros de control. En lugar de un valor final, se coloca una expresión lógica que repite el bucle mientras su resultado sea verdadero. Formato en C:

```
for (inicialización, condiciónIteración, incremento) {
    Sentencias del bucle
}
```

- *Inicialización*: Inicializa la variable de control del bucle.
- *condiciónIteración*: Expresión lógica que ejecuta el bucle mientras sea verdadera.
- *incremento*: Incrementa o decrementa la variable de control del bucle.

Ejemplo: Muestra un saludo 10 veces

```
# include <stdio.h>
int main () {
    int i;
    for (i=0; i<10; i++) {
        printf("Hola! \n");
    }
    return 0;
}
```

En el ejemplo, la variable de control es “i” de tipo entero, toma el valor inicial estipulado en el parámetro de iniciación, en este caso, 0. Por cada iteración del bucle incrementa su valor en 1. La repetición del bucle finaliza cuando “i” toma el valor 10. El bucle se repite mientras $i < 10$.

2.4.4. La estructura MIENTRAS (WHILE)

Esta estructura se utiliza cuando no se conoce de antemano la cantidad de veces que será necesario repetir un conjunto de instrucciones para solucionar el problema, o bien es conveniente que estas instrucciones sean repetidas hasta alcanzar una determinada condición. Por esta razón, a estos bucles se los denomina *bucles condicionales*.

La estructura MIENTRAS tiene la forma:

Seudo	Formato C
MIENTRAS condición HACER INSTRUCCIÓN [...] INSTRUCCIÓN FIN-MIENTRAS	while (condición-bucle) { sentencias; }

¿Cómo funciona? Al ejecutarse, lo primero que realiza es la **evaluación de la condición o expresión lógica**. Si el resultado es **Falso**, no se realiza ninguna acción y el programa prosigue en la siguiente sentencia después del bucle. Si el resultado es **Verdadero**, se ejecuta el cuerpo del bucle y se evalúa de nuevo la expresión. Este proceso se repite mientras la expresión lógica permanezca verdadera.

Reglas de funcionamiento:

1. La condición se evalúa antes de cada ejecución del bucle. Si la condición es verdadera, se ejecuta el bucle, y si es falsa, el control pasa a la instrucción siguiente al bucle.
2. Si la condición se evalúa a falso cuando se ejecuta el bucle por primera vez, el cuerpo del bucle no se ejecutará nunca. En este caso, se dice que se ha ejecutado 0 veces.
3. Mientras la condición sea verdadera, el bucle se ejecutará. Esto significa que el bucle se ejecutará indefinidamente a menos que “algo” en el interior del bucle modifique la condición haciendo que la condición pase a falso. Si la expresión nunca cambia de valor, entonces el bucle no termina nunca. En este caso, se trata de un bucle *infinito*. Deben evitarse estas situaciones.

Ejemplo: Escribir un programa que muestre el mensaje “Continuar avanzando”, hasta que se ingrese un dato que indique de que llegó a la meta.

```
#include <stdio.h>
int main () {
    char llegueAlaMeta = 'N';
    while (llegueAlaMeta == 'N') {
        printf("Continuar avanzando!\n");
        printf("Llegue a la meta? (N-S) ");
        fflush(stdin);
        scanf("%c", &llegueAlaMeta);
    }
    return 0;
}
```

Ejemplo de la salida resultante:

```
Continuar avanzando!
Llegue a la meta? (N-S) N
Continuar avanzando!
Llegue a la meta? (N-S) N
Continuar avanzando!
Llegue a la meta? (N-S) S
-----
Process exited with return value 0
Press any key to continue . . .
```

Uso de la indentación:

Para aumentar la legibilidad y claridad de un bucle MIENTRAS, el cuerpo del bucle se debe “sangrar” respecto de la palabra reservada MIENTRAS, tal como se muestra en el ejemplo anterior, de manera que quede visible el conjunto de instrucciones (bucle) que forma parte del MIENTRAS. Esta estrategia facilita la lectura y comprensión del código.

2.4.5. La estructura HACER-MIENTRAS (DO-WHILE)

La instrucción HACER-MIENTRAS se utiliza para especificar un bucle condicional que se ejecuta al menos una vez.

Seudo	Formato C
HACER <i>instrucción</i> [...] <i>instrucción</i> MIENTRAS <i>condición</i>	do { instrucción 1; instrucción 2; * } while (condicion)

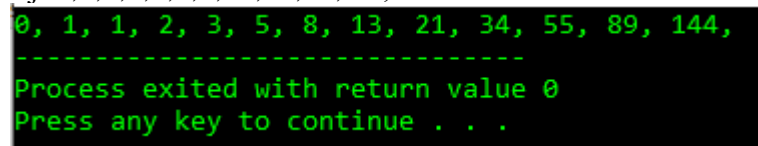
Reglas de funcionamiento:

1. La condición (expresión lógica) **se evalúa al final del bucle**, después de ejecutarse todas las sentencias.
2. Si la expresión lógica es verdadera, se vuelve a repetir el bucle y se ejecutan todas las sentencias.
3. Si la expresión lógica es falsa, se sale del bucle y se ejecuta la siguiente instrucción al final del **while**.
4. Si la condición se evalúa a falso cuando se ejecuta el bucle por primera vez, el cuerpo del bucle se ejecutará al menos una vez.

Ejemplo: Algoritmo para generar la serie de Fibonacci (*)

```
# include <stdio.h>
int main () {
    int a = 0;
    int b = 1;
    int c;
    printf("%d, ", 1, 2);
    printf("%d, ", a, b);
    do {
        c = (a + b);
        printf("%d, ", c);
        a = b;
        b = c;
    }
    while ((a+b)<200);
    return 0;
}
```

(*) En matemáticas, la serie de Fibonacci es una sucesión infinita de números naturales que comienza con los números 0 y 1, a partir de estos, cada término es la suma de los dos anteriores. Ej. 0,1,1,2,3,5,8,13,21,34, 55, ...



```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
-----
Process exited with return value 0
Press any key to continue . . .
```

2.4.6. Sentencias de salto en bucles**Interrumpir (Break)**

Esta instrucción se utiliza cuando se desea terminar un bucle en un lugar determinado del cuerpo del bucle sin esperar a que este termine de modo natural por su entrada o su salida. Esta instrucción corta el ciclo de ejecución por tanto debe ser utilizada con precaución.

Ejemplo: El programa suma los valores ingresados siempre que sean enteros positivos, con valor menor a 100. Si erróneamente se ingresa un valor negativo, el bucle termina, tal como se muestra en el siguiente pseudocódigo.

```
LEER número
DO
    IF número <= 0 INTERRUMPIR
        suma = suma + número
    LEER número
FIN-SI
WHILE número < 100
```

La instrucción Interrumpir sale del bucle DO y sigue con la instrucción siguiente a la instrucción WHILE.

Código C:

```
#include <stdio.h>
int main(){
    int numero;
    int suma=0;

    printf("Ingrese un numero >0 y <100 para sumar: ");
    scanf("%d",&numero);

    do {
        if ( numero < 0) {
            printf("\n --> numero incorrecto %d: ", numero);
            break;
        }
        suma = suma + numero;
        printf("Ingrese un numero >0 y <100 para sumar: ");
        scanf("%d",&numero);
    }while (numero <100);

    printf("\n\nsuma de numeros %d",suma);
    return 0;
}
```

Si se ingresan solamente datos correctos, la salida mostrará la suma de los números.

```
Ingrese un numero >0 y <100 para sumar: 40
Ingrese un numero >0 y <100 para sumar: 37
Ingrese un numero >0 y <100 para sumar: 120

suma de numeros 77
-----
Process exited with return value 20
Press any key to continue . . .
```

Si se ingresa un número incorrecto, un número negativo, la salida mostrará el mensaje de error sobre el valor incorrecto, y mostrará la suma de los números ingresados previamente.

```
Ingrese un numero >0 y <100 para sumar: 40
Ingrese un numero >0 y <100 para sumar: 37
Ingrese un numero >0 y <100 para sumar: -3

--> numero incorrecto -3:

suma de numeros 77
-----
Process exited with return value 20
Press any key to continue . . .
```

Compruebe, copiando y ejecutando el código ejemplo.

La sentencia **break** termina la ejecución de un bucle, o, en general de cualquier sentencia. Se utiliza para la salida de un bucle while o do-while, también se utiliza dentro de una sentencia switch, siendo éste su uso más frecuente.

Continuar (Continue)

Esta instrucción hace que el flujo de ejecución salte el resto del cuerpo del bucle para continuar con la siguiente iteración. Esta característica suele ser útil en algunas condiciones.

Ejemplo: Mostrar los números comprendidos entre 0 y 20 que no son múltiplos de 4.

Seudocódigo:

```

PARA i DESDE 0 HASTA 20
    SI (i mod 4 = 0) ENTONCES
        CONTINUAR
    FIN-SI
    ESCRIBIR i, ' ' // la constante " " separa los números en la salida
FIN-PARA

```

En este ejemplo, si el valor de **i** es múltiplo de 4, no escribe ese valor en la salida.

El resultado de este bucle será: 1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15, 17, 18, 19.

La sentencia CONTINUAR no afecta la cantidad de veces que se debe ejecutar el bucle.

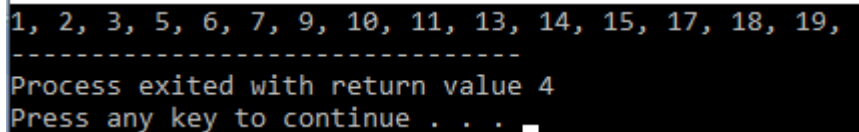
Código C:

```

#include <stdio.h>
int main(){
    int i;
    for (i = 0; i < 20; i++) {
        if (i % 4 == 0) {
            continue;
        }
        printf("%d, ", i);
    }
    return 0;
}

```

La salida resultante es la siguiente:



```

1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15, 17, 18, 19,
-----
Process exited with return value 4
Press any key to continue . . .

```

Compruebe, copiando y ejecutando este código.

2.4.7. Consideraciones a tener en cuenta en el diseño de los bucles:

El diseño correcto de los bucles es una tarea que requiere atención para lograr un programa eficiente. Para evitar errores se propone tener en cuenta las siguientes reglas (los ejemplos están codificados en C):

I. Inicializar variables:

Algunos lenguajes de programación inicializan las variables por defecto cuando no se les ha dado valores previamente mediante instrucciones de lectura (read) o instrucciones de asignación (=). Para evitar errores, se recomienda como buena práctica de programación inicializar las variables que serán usadas en el programa.

De igual modo, las variables que aparecen en el lado derecho de una sentencia de asignación ($X = Z + 1$) o en cualquier instrucción donde se espera encontrar un valor (contador $< N$), deben tener un valor inicial.

Ejemplo: Algoritmo que suma N cantidades ingresadas por teclado:

```
# include <stdio.h>
int main () {
    int suma, contador, cantidad, N;
    contador = 0;
    suma = 0;
    scanf ("%d", &N);
    while (contador < N) {
        scanf ("%d", &cantidad);
        suma = suma + cantidad;
        contador = contador + 1;
    }
    printf ("Valor de N ingresado: %d Total suma %d Cantidad de iteraciones %d \n", N, suma, contador);
    return 0;
}
```

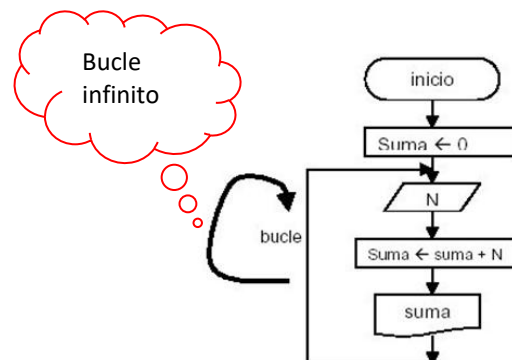
Las variables **suma** y **contador** se inicializan en 0 antes del bucle. En algunos lenguajes de programación, si no se les asigna un valor inicial la ejecución de la instrucción de asignación **suma = suma + cantidad** podrá generar un valor impredecible o cancelar por error en el tipo de dato.

II. Evitar bucles infinitos:

Algunos bucles no exigen terminación, por ejemplo, un sistema de monitoreo de alarma repetirá su algoritmo de detección hasta que el usuario desconecte el sistema de alarma. Sin embargo, lo más frecuente es que un bucle se diseñe para realizar un número finito de pasos hasta lograr el objetivo (ejemplo: calcular el promedio de notas de los alumnos). Un bucle infinito no intencionado debe ser evitado.

Por ejemplo, observemos el siguiente bucle:

```
# include <stdio.h>
int main () {
    int suma, contador, cantidad, N;
    contador = 0;
    suma = 0;
    while (contador != 10 ) {
        scanf ("%d", &cantidad);
        suma = suma + cantidad;
        contador = contador + 2;
    }
}
```



Este bucle se termina cuando **contador** = 10, después de realizar 5 iteraciones (**contador** se incrementa en 2).

Pero, si en lugar de incrementar **contador** en 2 se incrementa en 3, el contador nunca pasará por el valor 10, en consecuencia, el **bucle no se terminará nunca**.

Como regla general, conviene terminar un bucle con una condición de mayor o menor, evitar la verificación de igualdad o desigualdad. Hay que tener en cuenta que los datos numéricos reales se almacenan como cantidades aproximadas.

III. Finalización de los bucles:

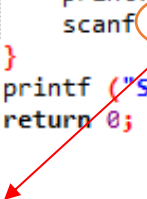
Si el programa está leyendo una lista de valores ó realizando una serie de cálculos en un bucle WHILE, se debe incluir **algún mecanismo de finalización del bucle**.

En general, existen dos métodos para controlar o terminar un bucle: bucles controlados por contador (bucles de conteo) y bucles condicionales.

a) Ingreso de valores desde el operador o usuario del programa

Este método consiste en preguntar al usuario si existen más entradas. Por ejemplo:

```
# include <stdio.h>
int main () {
    int suma, cantidad;
    char respuesta;
    respuesta = 'S';
    suma = 0;
    while (respuesta == 's' || respuesta == 'S') {
        printf ("ingrese un numero entero: \n");
        scanf ("%d", &cantidad);
        suma = suma + cantidad;
        printf ("Existen mas numeros en la lista? Ingrese una letra S o N \n");
        scanf ("%c", &respuesta);
    }
    printf ("Suma total de numeros %d \n ", suma);
    return 0;
}
```



En este caso, para el ingreso de un dato tipo char, se agrega previamente al especificador %c el carácter de control /n, para que tome el carácter de control que haya quedado en el buffer de entrada del scanf anterior, caso contrario, puede no ingresar correctamente el valor.

Este bucle se realizará mientras que el valor de la respuesta sea distinto de ‘S’ o ‘s’.

b) Uso de un valor “centinela” o valor de fin de dato:

Un *centinela* es un valor especial utilizado para indicar el final de una lista de datos. El valor elegido debe ser totalmente distinto de los posibles valores de la lista. Por ejemplo, si se trata de una lista de números positivos, se puede utilizar un valor negativo para indicar el valor final de la lista.

Por ejemplo:

```
# include <stdio.h>
int main () {
    int suma, numero;
    suma = 0;
    printf ("Ingrese un numero \n");
    scanf ("%d", &numero);
    while (numero > 0 ) {
        suma = suma + numero;
        printf ("Ingrese un numero \n");
        scanf ("%d", &numero);
    }
    printf ("Total suma %d \n", suma);
    return 0;
}
```

Si ejecutamos el código con esta lista de números: **7, 10, 2, 1, 6, -1**. El número **-1** se lee, pero no se suma y provoca la finalización del bucle.