

## CAPITULO II

### ANÁLISIS DE PROBLEMAS Y FORMULACIÓN DE ALGORITMOS

#### 1. INTRODUCCION

Aunque el proceso de diseñar programas es un proceso creativo, se pueden considerar una serie de fases o pasos comunes, que generalmente deben seguir todos los programadores.

La resolución de problemas con computadoras se puede dividir en tres fases:

- Análisis del problema
- Diseño del algoritmo
- Resolución del algoritmo en la computadora

El análisis y el diseño del algoritmo requiere la descripción del problema en subproblemas a base de *refinamientos sucesivos* y una herramienta de programación:

- Diagrama de flujo
- Pseudocódigo
- Codificación

Durante la tercera etapa se implementa este algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas obtenidas en las fases de análisis y diseño.

#### 2. ALGORITMO

Se deriva de la traducción al latín de la palabra árabe Alkhowarismi, nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX.

Un algoritmo es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

##### *2.1 Características del Algoritmo*

- Preciso, tiene que indicar el orden de realización en cada paso. definido, es decir, si el algoritmo se prueba dos veces, en estas dos pruebas,

se debe obtener el mismo resultado.

- Finito, es decir, que el algoritmo tiene que tener un número determinado de pasos.
- Debe producir un resultado en un tiempo finito.

#### Algunos ejemplos de algoritmos:

Ver una película

Buscar el videocasete de la película

SI el televisor y la video se encuentran apagados, encenderlos

Sacar el video del estuche

Introducirlo en la videocasetera

Tomar el control del televisor y la video

Dirigirme a el sofá

Ponerme cómodo

Disfrutar la película

Este pequeño algoritmo cumple con los requisitos descritos arriba, ya que cada paso precisa un orden y tiene un orden de pasos finitos. En este algoritmo aparece la palabra SI remarcada en mayúsculas, el uso de esta palabra la veremos mas adelante, cuando discutamos sobre el control del flujo del programa o estructuras de control.

Los algoritmos se pueden expresar por fórmulas, diagramas de flujo, y pseudocódigos conocidos como herramientas de programación. Está última representación es la mas utilizada por su sencillez y parecido a el lenguaje humano.

Para practicar estos conceptos se recomienda que escribieras algunos algoritmos de sucesos en tu vida cotidiana, como por ejemplo: encender el auto, ir al cine, preparar una torta, etc..

### **3. HERRAMIENTAS DE PROGRAMACIÓN**

Las herramientas de programación más utilizadas comúnmente para diseñar algoritmos son:

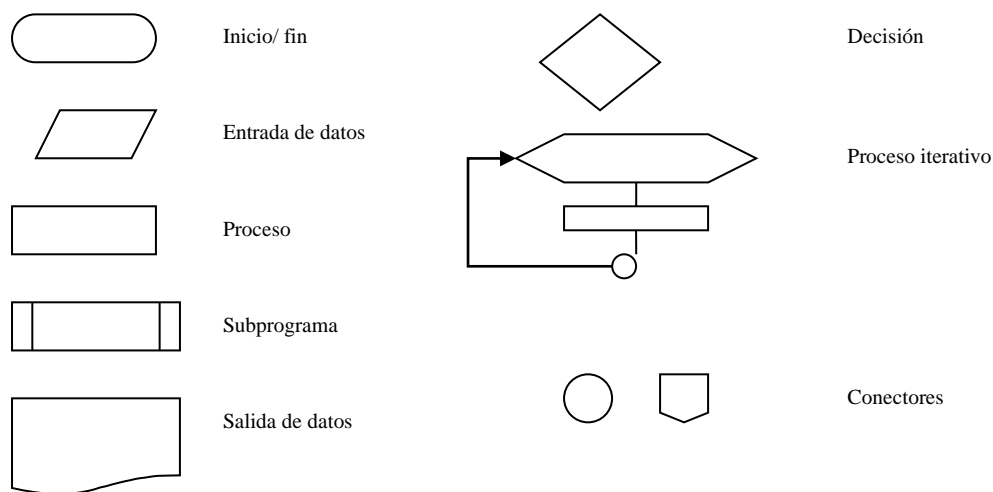
- Pseudocódigos
- Diagramas N-S
- Diagramas de flujo
- Codificación

### 3.1. Diagrama de flujo.

Es la representación gráfica de un algoritmo. Un diagrama de flujo debe reflejar :

- 1 el inicio del programa
- 2 las operaciones
- 3 la secuencia en que se realizan
- 4 el final del programa

Los signos más importantes de un diagrama de flujo son los siguientes:



### 3.2 Pseudocódigo

Permite escribir un algoritmo utilizando el lenguaje natural.

La lectura de datos permite asignar valores desde dispositivos hasta archivos externos en memoria, esto se denomina operación de entrada o lectura.

La operación de entrada en pseudocódigo se representa de la siguiente manera:

Leer (lista de variables)

Ejemplo:

Leer (A,B)

A medida que se realizan cálculos en el programa, se necesitan visualizar los resultados.

Esta se conoce como operación de escritura o salida.

Esta operación se representa en pseudocódigo de la siguiente manera:

Mostrar (lista de variables,"Mensaje") o Escribir (lista de variables,"Mensaje")

Ejemplo:

Mostrar ("La suma es : ", Sum)

En la instrucción de salida se pueden incluir además mensajes de texto y variables.

Para realizar una operación :

Asignar y la operación a realizar (la operación puede ser aritmética o de asignación)

Ejemplo:

Asignar B=5

Asignar suma = A + 10

### **Ejemplo: Suma de dos números ingresados por el usuario**

#### **Inicio \_Algoritmo Suma**

Declarar variables numero1, numero2 son enteros

**escribir** ("Introduce el primer número:")

**leer** (numero1)

**escribir** ("Introduce el segundo número:")

**leer** (numero2)

**Asignar** total= numero1 + numero2

**Escribir** ("El total es: ", total )

#### **Fin**

Esto debe visualizar en pantalla lo siguiente:

Introduce el primer número:

89

*//Intro presionado por el usuario*

Introduce el segundo número

1

*// Intro presionado por el usuario*

El total es: 90

### **3.3 Codificación**

La codificación es la traducción de un algoritmo en programa utilizando un lenguaje de programación

El lenguaje que se utilizara para implementación de los algoritmos es el lenguaje C++.

Para el ingreso de datos se utiliza :

cin >> variable;

(toda sentencia termina con ;)

Ejemplo :

cin>>A;      cin>>A>>B;    (leer varias variables se separa con los operadores )

Mostrar mensajes y/o variables se utiliza:

cout<<"mensaje"<<variable ;

Ejemplo:

cout<<"La suma es :"<< sum;

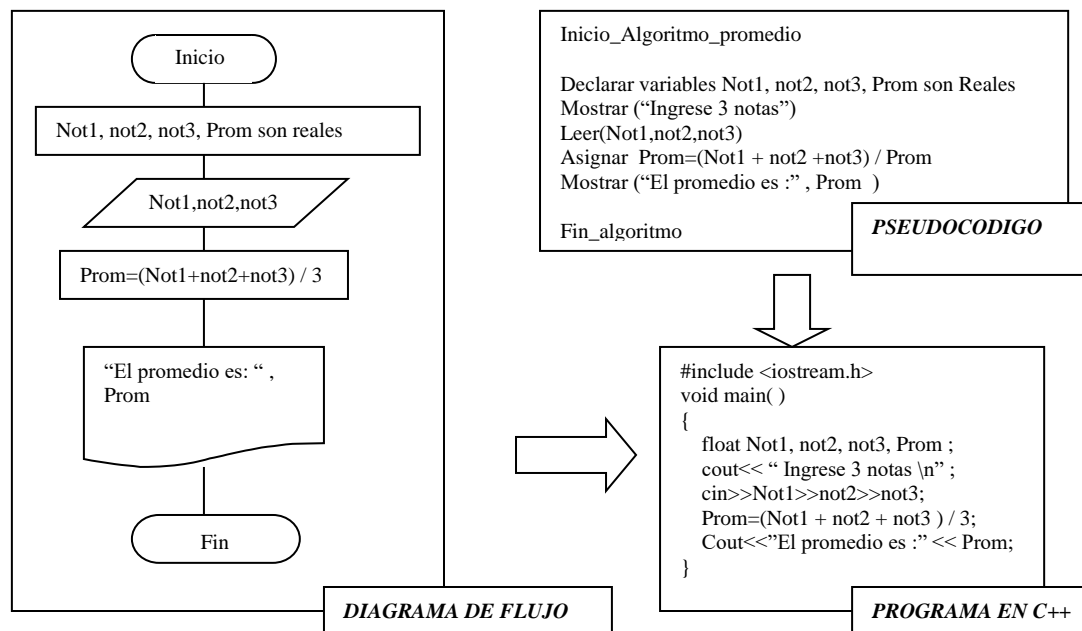
Todo mensaje se encierra entre comillas dobles y para mostrar lo que guarda la variable sin comillas .

Para realizar un cálculo aritmético (proceso):

$$\text{Sum} = A + B ;$$

## Ejemplo de las tres herramientas de programación

Calcular el promedio de 3 notas. Visualizar dicho promedio



## 2. TIPOS DE DATOS

Los diferentes objetos de información con los que un programa trabaja se denominan datos.

Todos los datos tienen un tipo asociados con ellos que nos servirá para poder conocer con que información trabajaremos. Es decir, cuando ingresemos el sueldo de un trabajador necesitamos que este contenga decimales, o al solicitar la edad de una persona está tiene que estar con números enteros, etc..

La asignación de tipos a los datos tiene dos objetivos principales:

- Detectar errores de operaciones aritméticas en los programas
- Determinar como ejecutar las operaciones

### 4.1 Tipos de Datos Comunes

Estos son los tipos de datos mas utilizados en los lenguajes de programación:

- Numéricos
- Caracteres (Texto)
- Lógicos

## Tamaños de los tipos fundamentales

Tipo	Tamaño
<code>bool</code> , <code>char</code> , <code>unsigned char</code> , <code>signed char</code> , <code>__int8</code>	1 byte
<code>__int16</code> , <code>short</code> , <code>unsigned short</code> , <code>wchar_t</code> , <code>__wchar_t</code>	2 bytes
<code>float</code> , <code>__int32</code> , <code>int</code> , <code>unsigned int</code> , <code>long</code> , <code>unsigned long</code>	4 bytes
<code>double</code> , <code>__int64</code> , <code>long double</code> , <code>long long</code>	8 bytes
<code>__int128</code>	16 bytes

## Tipos Numéricos

Dentro de estos tipos se puede hacer mención de los tipos enteros y reales o de coma flotante.

En C++ :

### ENTEROS

`int`  
`short`  
`long`  
`enum`

### REALES

`float`  
`double`  
`long double`

<i>TipodeDato</i>	<i>EspacioenMemoria</i>	<i>Rango</i>
<code>unsigned char</code>	8 bits	0 a 255
<code>char</code>	8 bits	-128 a 127
<code>short int</code>	16 bits	-32,768 a 32,767
<code>unsigned int</code>	32 bits	0 a 4,294,967,295
<code>int</code>	32 bits	-2,147,483,648 a 2,147,483,647
<code>unsigned long</code>	32 bits	0 a 4,294,967,295
<code>enum</code>	16 bits	-2,147,483,648 a 2,147,483,647
<code>long</code>	32 bits	-2,147,483,648 a 2,147,483,647
<code>float</code>	32 bits	3.4 x 10 <sup>-38</sup> a 3.4 x 10 <sup>+38</sup> (6 dec)
<code>double</code>	64 bits	1.7 x 10 <sup>-308</sup> a 1.7*10 <sup>+308</sup> (15 dec)
<code>long double</code>	80 bits	3.4 x 10 <sup>-4932</sup> a 1.1 x 10 <sup>+4932</sup>
<code>void</code>	sin valor	

## **Tipos Carácter**

Los tipos carácter se dividen también en caracteres ASCII, como por ejemplo: a A & \* , etc. El otro grupo de caracteres son los char o cadenas de caracteres, como por ejemplo: "Hola Mundo".

En C++:

```
char
```

## **Tipos Lógicos**

Los tipos lógicos solamente pueden tomar los valores verdadero o falso (true o false), también se puede usar los valores enteros (0,1) respectivamente.

## **3. IDENTIFICADORES**

Representan los nombres de los objetos de un programa (constantes, variables, tipos de datos, procedimientos, funciones, etc.). Es una secuencia de caracteres que puede ser de cualquier longitud, aunque tiene ciertas reglas que hay que seguir, las cuales son: Debe comenzar con una letra y no puede contener espacios en blanco. Letras, dígitos y caracteres subrayados ("\_") están permitidos después del primer carácter. En síntesis, un **identificador** es un método para nombrar a las celdas de memoria en la computadora, en lugar de memorizarnos una dirección de memoria. Se utilizan para nombrar variables, constantes, procedimientos y funciones.

## **4. CONSTANTES**

Las constantes son valores que no pueden cambiar en la ejecución del programa. Recibe un valor en el momento de la compilación del programa y este no puede ser modificado.

En C++ una constante puede ser un número, un carácter o una cadena de caracteres

Ejemplo:

```
const int A=100
```

## **5. VARIABLES**

Las variables son valores que se pueden modificar durante la ejecución de un programa. Al contrario de las constantes que no pueden ser cambiados.

Para declarar una variable en C++ :

```
Tipo_de_dato nombre_variable
```

Ejemplo :

```
int A,b;
```

## **6. EXPRESIÓN .**

Una expresión es una variable, constante o una formula a evaluar

X= 5                                      Es una constante

Area = (base \*altura ) / 2        ES una fórmula

Z = A                                      Es una variable

Operadores aritméticos básicos

OPERADOR	C++	SIGNIFICADO	EJEMPLO	RESULTADO
+		Suma	A + b	Suma de a y b
-		Resta	A - b	Resta de a y b
*		Multiplicación	A * b	Producto de a y b
/		Division con decimales	A / b	Cociente de a y b
Div	/	Division entera	A div b //variables deben ser de tipo entero	Cociente entero de a y b
mod	%	Residuo	A mod b	Resto de a y b

## 8. 1 Reglas de Evaluación

Todas las subexpresiones entre paréntesis se evalúan primero. Las subexpresiones entre paréntesis anidados se evalúan de adentro hacia afuera, es decir, que el paréntesis mas interno se evalúa primero.

- Prioridad de Operaciones: Dentro de una misma expresión o subexpresión, los operadores se evalúan en el siguiente orden:

\* , / , div y mod        primero  
+, -                                      ultimo

- Los operadores en una misma expresión o subexpresión con igual nivel de prioridad se evalúan de izquierda

Ejemplo:

1.        5 + 1\*10  
          5 + 10  
          15
2.        3 + 5 \* (10 - (3 + 2))  
          3 + 5 \* (10 - 5)  
          3 + 5 \* 5  
          3 + 25  
          28
3.        4 + 9 mod 2  
          4 + 1  
          5

## 9. EXPRESIONES LÓGICAS



En los programas con frecuencia debemos enfrentarnos con situaciones en las que se deben proporcionar instrucciones alternativas que pueden o no ejecutarse, dependiendo de los datos de entrada, reflejándose el cumplimiento o no de una determinada condición.

## 9.1 Operadores relacionales

En ocasiones en los programas se necesitan realizar comparaciones entre distintos valores, esto se realiza utilizando los operadores relaciones, los cuales se listan a continuación:

Operador	C++	Significado
<	<	Menor que
>	>	Mayor que
≤	<=	Menor o igual que
≥	>=	Mayor o igual que
=	==	Igual a
<>	!=	Distinto

Una expresión lógica es una expresión que puede ser verdadera o falsa

Ejemplo:

$(2 + 2 * 5) < > 6 + (5 - 2)$

$(2 + 10) < > 6 + 3$

12 < > 9

Verdadero

## 9.2 Operadores Lógicos

Las expresiones lógicas pueden combinarse para formar expresiones más complejas utilizando los operadores lógicos: and , or y not

And			Or			Not	
Exp1	Expr2		Exp1	Expr2		Not	Exp1
V	V	V	V	V	V	V	F
V	F	F	V	F	V	F	V
F	V	F	F	V	V		
F	F	F	F	F	F		
&&						!	
						C++	

Ejemplo

A tiene el valor de 5, B tiene el valor de 3

$((a * 2 > b - 3) \text{ or } (a > b - 1)) \text{ and } (b < 1)$

$((10 > 0) \text{ or } (5 > 2)) \text{ and } (3 < 1)$

( V or V ) and F

V and F

F

## 10. ALGORITMOS SECUENCIALES

### 10.1 Aplicaciones

1. Escribir un algoritmo para convertir metros a pies y pulgadas

1 Metro = 39.27 pulgadas  
1 pie = 12 pulgadas

#### **Análisis**

Variables de entrada

Metro =?

Variables de salida

Pies y pulgadas

1. multiplicar número de metros por 39.27
2. dividir el resultado anterior por 12

#### **Diseño**

##### **Pseudocodigo**

Inicio\_Algoritmo\_metros

Declarar variables metro, pulgadas, pies son reales

Mostrar("ingrese la cantidad de metros a convertir")

Leer(metro)

Asignar  $pulgadas = metro * 39.27$

Asignar  $pies = pulgadas / 12$

Mostrar ("el equivalente de", metro, "metros es :")

Mostrar(pulgadas, "pulgadas y ", pies, "pies")

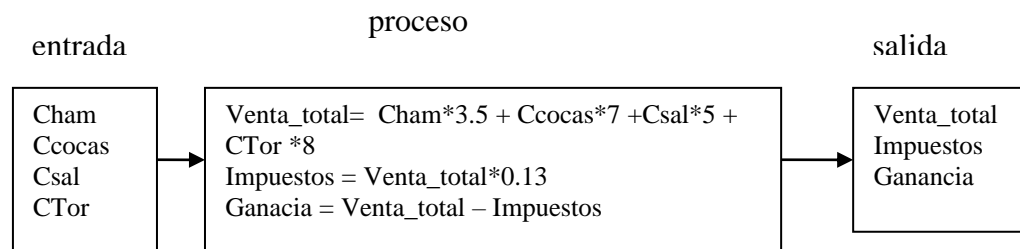
Fin\_alg.

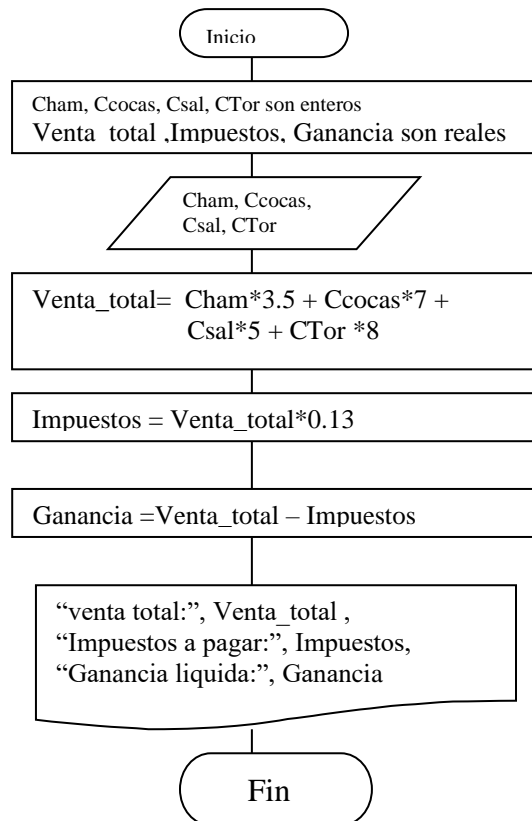
2. El menú de un restaurante rápido es :

Hamburguesa 3.5 Bs.  
Coca cola 7 Bs.  
Salchichas 5 Bs.  
Torta 8 Bs.

Se desea un algoritmo que calcule las ventas totales al final del día, así como los impuestos a pagar (13%)

#### **Análisis**





**DIAGRAMA DE FLUJO**

Las librerías a utilizar para desarrollar el programa son :

- La librería <iostream.h> se utiliza para la función cin(Leer) y cout (mostrar)
- La librería <conio.h> se utiliza para la función clrscr( ) y el getch( )

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

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

```
int main( )
```

```
{
```

```
    // declaracion de variables
```

```
    int Cham, Ccocas, Csal, CTor ; // variables enteras
```

```
    float Venta_total, Impuestos, Ganancia; // variables reales
```

```
    // muestra el mensaje por pantalla
```

```
    cout << "ingrese las cantidades vendidas de cada producto del restaurant \n";
```

```
    cin>> Cham>> Ccocas>> Csal>> CTor; // lee datos para las variable, se debe ingresar por teclado
```

```
    // realizar cálculos aritméticos para encontrar la venta total, impuestos y ganancia Liquida del dia
```

```
    Venta_total= Cham*3.5 + Ccocas*7 +Csal*5 + CTor *8 ;
```

```
    Impuestos = venta_total*0.13 ;
```

```
    Ganacia = Venta_total – Impuestos;
```

```
    Clrscr(); // limpia la pantalla
```

```
    // Muestra los resultados obtenidos por pantalla
```

```
    cout<<"Venta total :"<< Venta_total;
```

```
    cout<< "\n Impuestos a pagar :"<<Impuestos;
```

```
    cout <<"\n Ganancia liquida :"<<Ganacia;
```

```
    getch ( ) ; // detiene la ejecución del programa hasta que se presione una tecla
```

```
}
```

El "\n" hace que sea sin de línea es decir que muestre el mensaje en la línea siguiente

## EJEMPLOS DE PROGRAMAS

### 1) ENTRADA Y SALIDA

```
#include <stdio.h>

int main()
{
    int num;
    printf( "Introduce un número " );
    scanf( "%i", &num );
    printf( "Has tecleado el número %i\n", num );
    int a, b, c;
    printf( "Introduce tres números: " );
    scanf( "%i %i %i", &a, &b, &c );
    printf( "Has tecleado los números %i %i %i\n", a, b, c );
}
```

### 2) NUMEROS

```
#include <iostream>

using namespace std;
int main()
{
    int numero;
    cout<<"Introduzca un numero entero: ";
    cin >> numero;
    numero=numero+10;
    cout<<"El numero encrimentado en 10      es "<< numero << endl;
}
```

### 3.BOOLEAN

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
    bool bandera;
    bandera=true;
    if (bandera)
        cout<<"es verdadero ";
    printf("Adios");
}
```

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

### OTRO

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
    bool bandera;
    bandera=true;
    if (bandera)
        cout<<"es verdadero ";
    printf("Adios");
}
```

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

## METODOLOGIAS DE PROGRAMACION

## 3.1 CONCEPTOS BÁSICOS EN PROGRAMACIÓN.

**Programar** es escribir un código en un lenguaje de programación con las sentencias apropiadas para que se ejecuten las operaciones necesarias para un propósito determinado. Los programas son *códigos* diseñados, además de ser **ejecutados**, para ser **leídos**, **depurados**, **actualizados**, y **corregidos**. Excepto la primera de estas operaciones, el resto van a ser realizadas por una *persona* con ciertos conocimientos en la materia. Un **buen programa**, además de **correcto** (que realice las operaciones para las que fue diseñado) y **eficiente** (que emplee los menos recursos posibles; espacio y tiempo), debe facilitar las otras operaciones a realizar sobre él. El **estilo** de programación está constituido por aquellas normas ideadas para facilitar las operaciones a realizar sobre el código.

## 3.2 FASES DE DESARROLLO DE UN PROGRAMA.

Una buena **planificación** de las tareas a realizar para desarrollar un programa favorece el éxito en la implementación. La planificación debe estar basada en el establecimiento de fases. Las **fases**, tanto para realizar programas sencillos como para llevar a cabo proyectos de construcción de aplicaciones informáticas son las siguientes :describir el programa, escribirlo, verificar su comportamiento, depurar su funcionamiento y mantenerlo. Estas fases consisten en los siguiente:

- **Describir un programa.**

Describir un programa consiste en expresar el propósito del programa y los pasos a seguir. El resultado se denomina **pseudocódigo** porque está expresado un lenguaje intermedio entre el lenguaje natural y un lenguaje de programación. Los pseudocódigos son de distinto nivel según el detalle en la especificación de las operaciones a realizar. Un pseudocódigo de alto nivel será muy parecido al lenguaje natural y uno de bajo nivel será muy parecido a un programa de ordenador

- **Escribir un programa.**

Una vez establecido el pseudocódigo que describe un programa, escribirlo consiste en **traducir** el pseudocódigo en un programa correcto en el lenguaje de programación correspondiente.

- **Verificar un programa.**

Verificar un programa consiste en comprobar que realiza las operaciones para las que fue concebido. Para verificar el programa hay que diseñar y ejecutar un conjunto de pruebas para comprobar que su ejecución realiza las operaciones previstas. La verificación el programa debe complementarse con la **validación**, consistente en someterlo a las condiciones reales en las que va a actuar para garantizar una adecuada implantación definitiva.

- **Depurar un programa.**

Para depurar un programa y conseguir optimizar su rendimiento hay que realizar los

ajustes necesarios para garantizar el máximo aprovechamiento de los recursos y evitar los resultados erróneos. Para esta tarea se han diseñado herramientas específicas denominadas *depuradores*.

- ***Mantener un programa.***

Para el adecuado mantenimiento de un programa en un entorno específico es necesario introducir las modificaciones necesarias para adaptarse a las alteraciones en los *requerimientos* a los que está sometido.

### **3.3. METODOLOGÍAS DE PROGRAMACIÓN**

Las **metodologías** de la programación establecen las *herramientas formales* para diseñar la solución de un problema mediante un programa. Algunas de estas metodologías y sus herramientas son la programación estructurada y las estructuras básicas, la modularidad, la abstracción de datos, el diseño descendente, la recursividad, la programación orientada a objetos, la programación orientada a eventos, etc.

#### ***3.3.1 Programación estructurada***

La programación estructurada (en adelante simplemente PE ), es un estilo de programación con el cual el programador elabora programas, cuya estructura es la más clara posible, mediante el uso de tres estructuras básicas de control lógico, a saber :

- a. SECUENCIA.
- b. SELECCIÓN.
- c. ITERACIÓN.

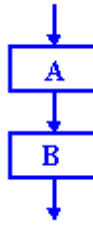
Un programa estructurado se compone de funciones, segmentos, módulos y/o subrutinas, cada una con una sola entrada y una sola salida. Cada uno de estos módulos (aún en el mismo programa completo), se denomina *programa apropiado* cuando, además de estar compuesto solamente por las tres estructuras básicas, tiene sólo una entrada y una salida y en ejecución no tiene partes por las cuales nunca pasa ni tiene ciclos infinitos.

La PE tiene un teorema estructural o teorema fundamental, el cual afirma que cualquier programa, no importa el tipo de trabajo que ejecute, puede ser elaborado utilizando únicamente las tres estructuras básicas ( secuencia, selección, iteración ).

##### ***3.3.1.1 Definición de las estructuras básicas de control lógico***

###### **a) SECUENCIA**

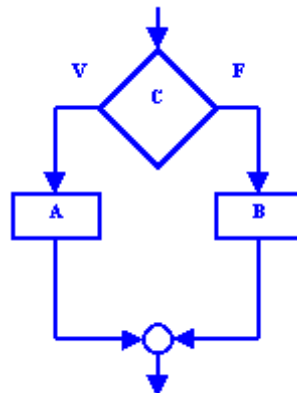
Indica que las instrucciones de un programa se ejecutan una después de la otra, en el mismo orden en el cual aparecen en el programa. Se representa gráficamente como una caja después de otra, ambas con una sola entrada y una única salida.



Las cajas A y B pueden ser definidas para ejecutar desde una simple instrucción hasta un módulo o programa completo, siempre y cuando que estos también sean programas apropiados.

### **b) SELECCIÓN**

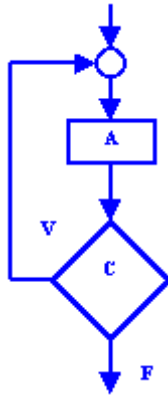
También conocida como la estructura SI-CIERTO-FALSO, plantea la selección entre dos alternativas con base en el resultado de la evaluación de una condición o predicado; equivale a la instrucción IF de todos los lenguajes de programación y se representa gráficamente de la siguiente manera :



En el diagrama de flujo anterior, C es una condición que se evalúa; A es la acción que se ejecuta cuando la evaluación de este predicado resulta verdadera y B es la acción ejecutada cuando indica falso. La estructura también tiene una sola entrada y una sola salida; y las funciones A y B también pueden ser cualquier estructura básica o conjunto de estructuras.

### **c) ITERACIÓN**

También llamada la estructura HACER-MIENTRAS-QUE, corresponde a la ejecución repetida de una instrucción mientras que se cumple una determinada condición. El diagrama de flujo para esta estructura es el siguiente :



Aquí el bloque A se ejecuta repetidamente mientras que la condición C se cumpla o sea cierta. También tiene una sola entrada y una sola salida; igualmente A puede ser cualquier estructura básica o conjunto de estructuras.

### 3.3.1.2 VENTAJAS DE LA PROGRAMACIÓN ESTRUCTURADA

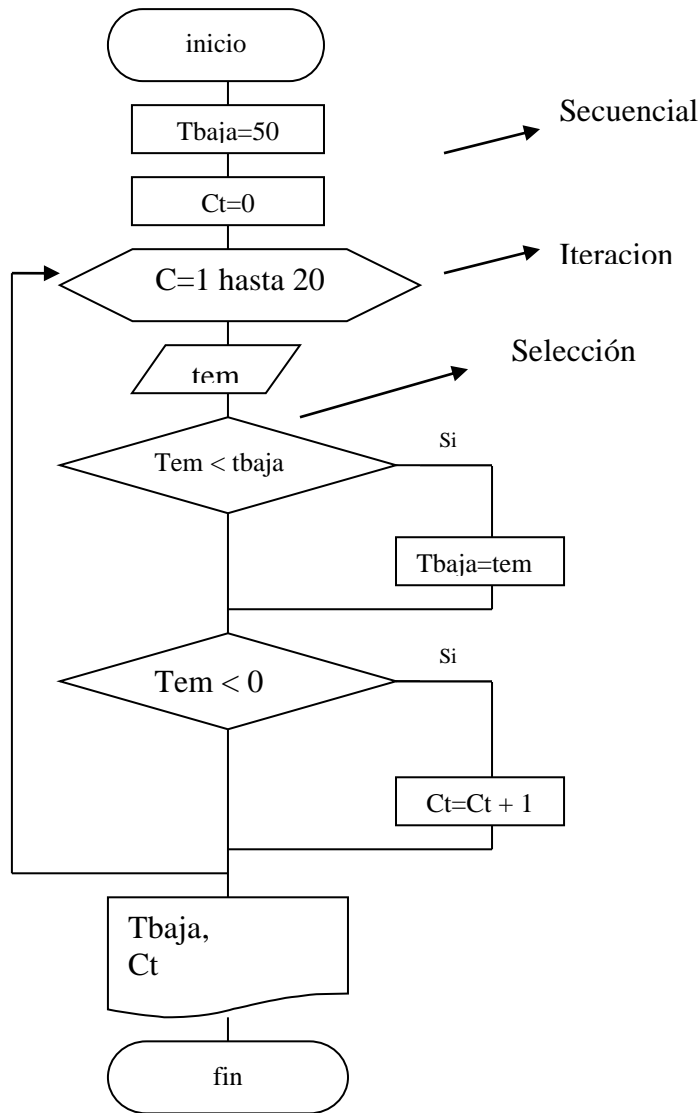
Con la PE, elaborar programas de computador sigue siendo una labor que demanda esfuerzo, creatividad, habilidad y cuidado. Sin embargo, con este nuevo estilo podemos obtener las siguientes ventajas :

1. Los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación. La estructura del programa es más clara puesto que las instrucciones están más ligadas o relacionadas entre si, por lo que es más fácil comprender lo que hace cada función.
2. Reducción del esfuerzo en las pruebas. El programa se puede tener listo para producción normal en un tiempo menor del tradicional; por otro lado, el seguimiento de las fallas o depuración (debugging) se facilita debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir más fácilmente.
3. Reducción de los costos de mantenimiento.
4. Programas más sencillos y más rápidos.
5. Aumento en la productividad del programador.
6. Se facilita la utilización de las otras técnicas para el mejoramiento de la productividad en programación.
7. Los programas quedan mejor documentados internamente

Ejemplo : En el siguiente ejemplo se puede ver la 3 estructuras básicas : secuencia , selección e iteración

Se dispone de 20 temperaturas tomadas a las 8 horas en la ciudad de Cochabamba durante el mes de abril. Se desea saber cuantas son inferiores a 0° grados y cual es la temperatura mas baja. (Diagrama de flujo)





### 3.3.2 Programación Modular

Uno de los métodos fundamentales para resolver un problema es dividirlo en problemas mas pequeños, llamados *subproblemas*, en referencias sucesivas.

Estos problemas a su vez pueden ser divididos repetidamente en problemas mas pequeños hasta que los problemas mas pequeños puedan ser solucionados.

Esta técnica de dividir el problema principal en subproblemas se denomina frecuentemente *divide y vencerás*. El método de diseño se denomina diseño descendente, debido a que se comienza en la parte superior con un problema general y se diseñan soluciones específicas

a sus subproblemas.

El problema principal se resuelve con el programa principal (también llamado conductor del programa), y los subproblemas (módulos) mediante subprogramas: *procedimientos* y *funciones*.

Un subprograma realiza una tarea concreta que se describe con una serie de instrucciones.

Los procedimientos y funciones son similares, aunque presentan notables diferencias entre ellos:

Las funciones normalmente, devuelven un solo valor a la unidad de programa (programa principal u otro subprograma) que los referencia o llama.

Los procedimientos pueden devolver cero, uno o varios valores. En el caso de no devolver ningún valor, realizan alguna tarea tal como alguna operación de entrada/salida.

A un nombre de procedimiento no se puede asignar un valor, y por consiguiente ningún tipo está asociado con un nombre de procedimiento.

Una función se referencia utilizando su nombre en una instrucción (de asignación o expresión matemática), mientras que un procedimiento se referencia por una llamada o invocación al mismo.

Ejemplo: Leer el radio de un círculo y calcular e imprimir su superficie y longitud.

- **Análisis**  
Especificaciones de Entrada  
Radio: Real  
Especificaciones de Salida  
Superficie: Real  
Longitud: Real

```
#include <iostream.h>
void superficie(float Radio ) // función no retorna ningún valor
{
    float S;
    const float pi=3.141592;
    S = pi * Radio * Radio;
    cout<<"la superficie es "<<S;
}
Float longitud(float radio ) //función retorna un valor la longitud
{
    const float pi = 3.141592,L;
    L=- 2 * pi * Radio;
    Return L;
}
```

```

void main( ) //programa principal
{
    float radio;
    cout<<"ingrese el radio";
    cin>>radio;
    superficie(radio); // llama la function superficie
    // llama la function longitud y el resultado que devuelve la funcion guarda en la variable long
    long=longitud(radio)
    cout<<"la longitud es"<<long;
}

```

### 3.3.3 Programación Orientada a Objetos

La **modularidad**, aplicada a la **abstracción** de procedimientos y de datos, junto con la filosofía de **ocultación** de información, conducen a la **programación orientada a objetos**. Esta metodología está basada en la combinación de **módulo procedimental** y **tipo abstracto de dato** para dar lugar al concepto de **objeto**. La abstracción o modularidad procedimental permite contemplar un programa que comprende diversos subprogramas o **módulos** sin necesidad de conocer los procedimientos que ejecutan esos subprogramas ni su implementación; sólo es necesario conocer su nombre y las características de sus parámetros. La abstracción o modularidad de los datos permite contemplar el uso de un **tipo de datos** sin conocer como se almacenan en la correspondiente estructura de datos ni cómo se implementan las operaciones que definen el tipo abstracto de datos. La **ocultación** de la información no sólo evita detalles ocultos innecesarios al nivel considerado sino que impide que otros elementos (módulos o TAD) puedan tener acceso a ellos.

Las **características** fundamentales de los objetos son el **encapsulamiento**, la **herencia** y el **polimorfismo**. El **encapsulamiento** es la combinación de datos y operaciones sobre esos datos. La **herencia** permite transmitir propiedades de un objeto a otros denominados descendientes. El **polimorfismo** permite decidir durante la ejecución de un programa la función a ejecutar sin necesidad de que esté previamente establecida.

Ejemplo : El siguiente ejemplo muestra la clase alumno con sus atributos y métodos

```

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

class alumno    //clase alumno
{
    private: // especificadores de acceso publico, privado y protegido
    //características o propiedades de clase alumno
    char cod[5];
    char nombre[10];
    char carrera[15];
    float promedio;

    public:
    void leer(char c[5], char n[10],char car[15]) // metodo de la clase alumno
}

```

```

{
strcpy(cod,c);
strcpy(nombre,n);
strcpy(carrera,car);
promedio=0;
}
void mostrar( ) //metodo de la clase alumno
{
cout<<"\n CODIGO DEL alumno : "<<cod;
cout<<"\n Nombre y apellido : "<<nombre;
cout<<"\n carrera      : "<<carrera;
cout<<"\n promedio : "<<promedio;

}

```

## ESTRUCTURAS DE CONTROL

### 4.1 INTRODUCCION

Por lo regular en un programa los enunciados son ejecutados uno después del otro, en el orden en que aparecen escritos. Esto se conoce como ejecución secuencial. Sin embargo, existen enunciados que le permiten al programador especificar que el enunciado siguiente a ejecutar pueda ser otro diferente al que sigue en secuencia. Esto se conoce como transferencia de control.

Las estructuras de control se clasifican en :

- Estructuras de control selectivas o de decisión
- Estructuras de control repetitivas o iterativas

### 4.1 ESTRUCTURAS DE CONTROL SELECTIVAS O DE DECISION

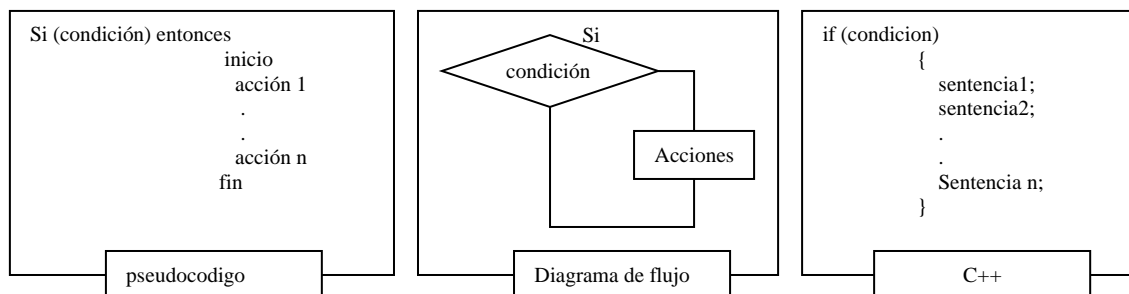
Las estructuras selectivas o de decisión se utilizan para tomar decisiones lógicas dependiendo si es verdadera o falsa . Se clasifican en los siguientes tipos:

- Decisión o alternativa simple
- Decisión o alternativa doble
- Decisión o alternativa multiple

#### 4.2.1 Decisión o alternativa simple : Si..entonces (if)

Dado que las expresiones lógicas toman el valor verdadero y falso, se necesita una sentencia de control para la toma de decisiones, cuando se desea ejecutar una acción si una expresión es verdadera o falsa.

Para ello utilizaremos la sentencia de selección if (si).



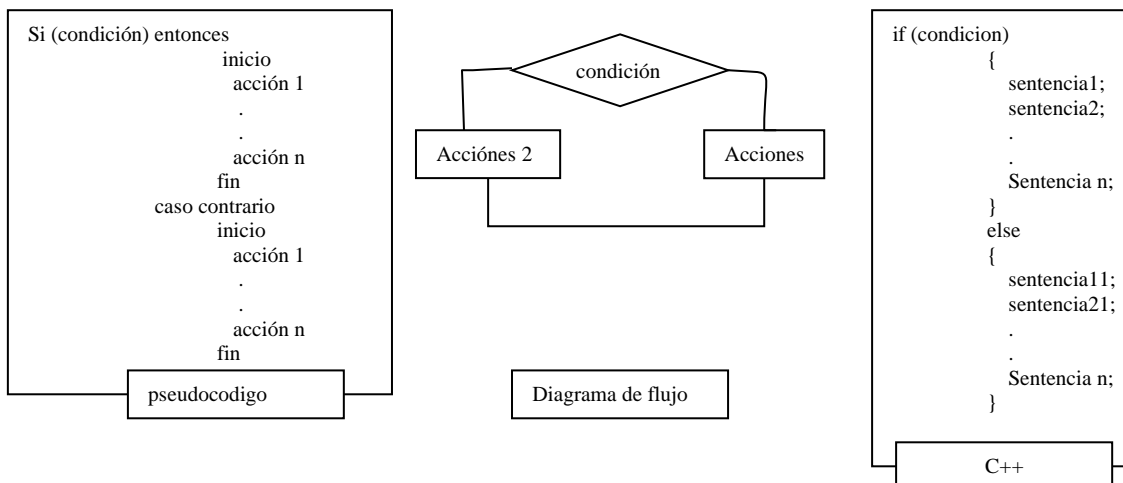
Cuando la expresión lógica contenida por los paréntesis es verdadera, se ejecutan las instrucciones dentro de la estructura de selección, cuando es falsa, el programa ignora la estructura y se sigue ejecutando la instrucción siguiente a la estructura de control.

Ejemplo: Calcular el salario a cobrar de un trabajador sabiendo que por día gana 100Bs. Si el trabajador trabaja mas de 4 días tiene un bono de 10Bs por día extra

<pre> Inicio _algoritmo-salario   Declarar dias, Dias_extra,Extra   son enteros   Declarar salario es real   Declarar constante gana_dia es entero   Gana_dia=100   Leer(dias)   Salario=dias * gana_dia   Si (dias&gt;4) entonces     inicio       Dias_extras= dias - 4       Extra=Dias_extra * 10       Salario=Extra + salario     Fin   Mostrar("el trabajador gana :", Salario) Fin_algoritmo </pre>	<pre> #include&lt;iostream.h&gt; Void main( ) {   int dias, Dias_extra,Extra;   const int  gana_dia=100;   cout&lt;&lt;"ingrese los dias que trabaja";   cin&gt;&gt;dias;   Salario=dias * gana_dia;   if (dias&gt;4)   {     Dias_extras= dias - 4;     Extra=Dias_extra * 10;     Salario=Extra + salario;   }   cout&lt;&lt;"el trabajador gana :"&lt;&lt; Salario } </pre>
---	--

#### 4.2.2 Decisión o alternativa doble : si/caso contrario (if/else)

La estructura de selección si/ caso contrario permite que el programador especifique la ejecución de una acción distinta cuando la condición es falsa.



Por ejemplo, el enunciado en seudocódigo:

```

Si (nota >= 60) entonces
    escribir ("Aprobado")
caso contraio
    escribir ("Reprobado")

```

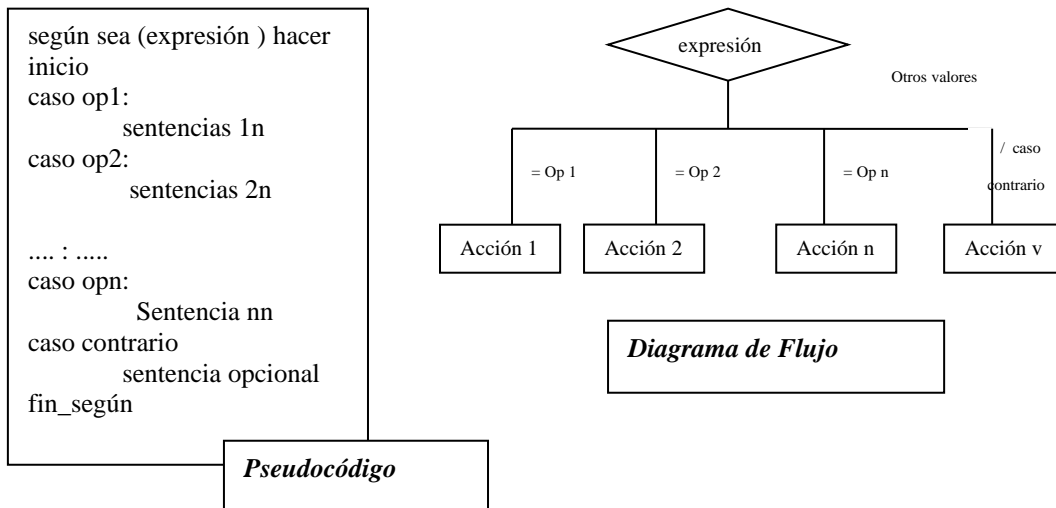
En C++

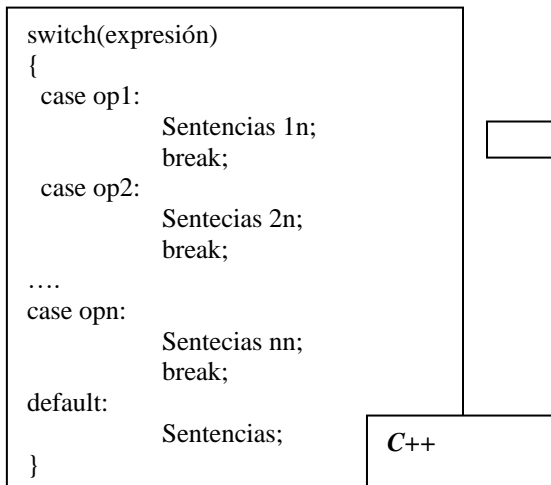
```
if (nota >=60)
    cout<<"Aprobado";
else
    cout<<"Reprobado";
```

Imprime "Aprobado", si la calificación del alumno es mayor o igual a 60, e imprime "Reprobado" si la calificación es menor que 60. En cualquiera de los casos, después de haber impreso alguno de los mensajes, el programa ejecutará el enunciado siguiente

#### 4.2.3. Decisión o alternativa múltiple : Según Sea (Case)

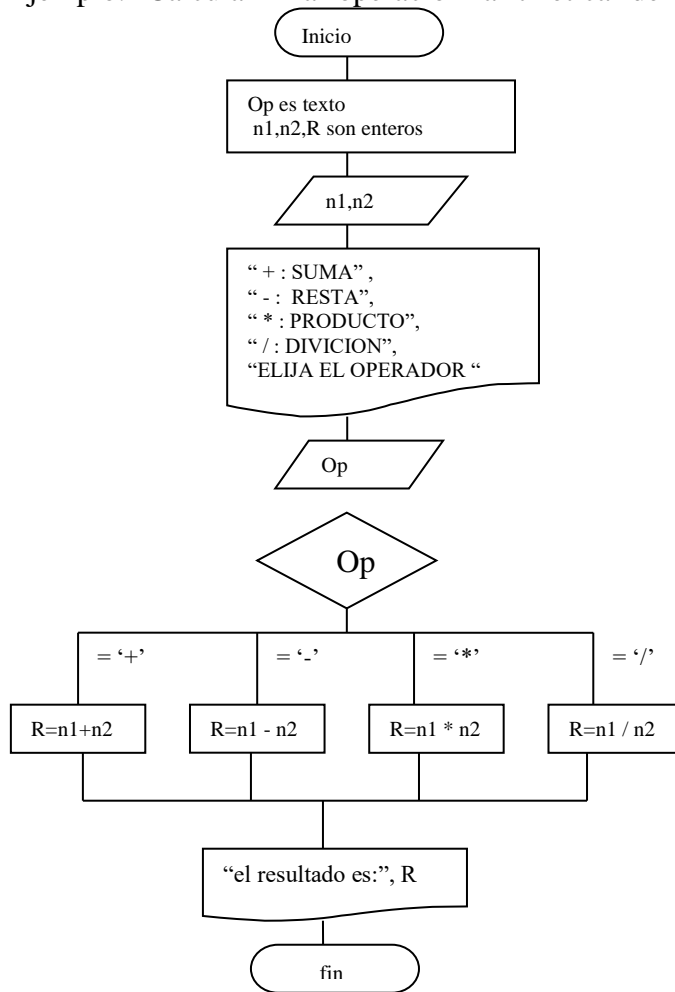
Esta sentencia se utiliza para elegir entre diferentes alternativas. Esta se compone de varias sentencias simples, cuando se ejecuta, una y solo una de las sentencias simples se selecciona y ejecuta. La sintaxis es la siguiente:





La expresión a evaluar debe ser del tipo de dato entero o de un solo carácter ("A", "+"), no acepta datos reales ni cadenas

Ejemplo: Calcular la operación aritmética de dos números ingresados por teclado



```

#include <iostream.h>
Void main()
{
  char op;
  int n1,n2,R;
  cout<<"Ingrese dos numeros\n";
  cin>>n1>>n2;
  cout<<" + : SUMA \n ";
  cout<<" - : RESTA\n";
  cout<<" * : PRODUCTO \n";
  cout<<" / : DIVISION\n";
  cout<<"ELIJA EL OPERADOR\n";
  cin>>Op;
  switch ( Op )
  {
    case '+':
      R=n1+n2;
      break;
    case '-':
      R=n1-n2;
      break;
    case '*':
      R=n1*n2;
      break;
    case '/':
      R=n1/n2;
      break;
    default:
      cout<<"no existe el operador";
  }
}

```



#### 4.2.4 *Sentencias Selectivas Anidadas*

Dentro de las sentencias que figuran dentro de una sentencia if, pueden colocarse también otras sentencias selectivas. De esta manera:

Supongamos que deseamos imprimir en pantalla la nota de un alumno, clasificándolo en "aprobado", "no aprobado", y "deficiente". El algoritmo quedaría de esta manera.

```
si (nota >= 60) entonces
    escribir ("aprobado")
caso contrario
    si (nota < 60) and (nota >= 30) entonces
        escribir ("no aprobado")
    caso contrario
        si (nota < 30) entonces
            escribir ("deficien
```

```
En C++
    if (nota >= 60)
        cout<<"aprobado";
    else
        if ( (nota < 60) && (nota >= 30))
            cout<<"no aprobado";
        else
            if (nota < 30)
                cout<<"deficiente";
```

### Ejemplos

```
#include <iostream>
using namespace std;
int main()
{
    bool bandera;
    int nota,expresion;
    bandera=true;
    /*simple */
    if (bandera)
        cout<<"es verdadero "<<endl;

    /*doble*/
    nota=55;
    if (nota >=60)
        cout<<"Aprobado"<<endl;
    else
        cout<<"Reprobado"<<endl;

    /*multiple*/
```

```

expresion=1;
switch(expresion)
{
    case 1:
        cout<<"UNO"<<endl;;
        break;
    case 2:
        cout<<"DOS"<<endl;;
        break;

    case 3:
        cout<<"TRES"<<endl;
        break;
    default:
        cout<<"POR DEFECTO"<<endl;
}

/*anidados */
nota=30;
if (nota >=60)
    cout<<"Aprobado"<<endl;

else
    cout<<"Reprobado"<<endl;
    if(nota<40)
        cout<<"deficiente"<<endl;

system("PAUSE");
}

```

## ESTRUCTURAS DE CONTROL REPETITIVAS O ITERATIVAS

Las estructuras de control repetitivas son aquellas en las que una sentencia o grupos de sentencias se repiten muchas veces dependiendo de una condición que puede ser evaluada al principio o al final del grupo de sentencias.

Una estructura de control que permite la repetición de una serie determinada de sentencias se denomina bucle (lazo o ciclo). El cuerpo del bucle contiene las sentencias que se repiten.

La acción o acciones que se repiten en un bucle se denominan el cuerpo del bucle, y cada repetición del cuerpo del bucle se denomina iteración.

Se clasifican en:

- Estructura Para o Desde
- Estructura Mientras
- Estructura Repetir

Estas estructuras de control repetitivas necesitan contadores y acumuladores los cuales son manejados dentro de una estructura repetitiva.

### 4.1.1 Contador

Un contador es una variable cuyo valor se incrementa o decrementa en una cantidad fija en cada iteración. Los contadores siempre son de tipo entero normalmente se usan para contar los sucesos o acciones internas del bucle

$$\text{cont} = \text{cont} \pm \text{constante}$$

**Ejemplo:** El nombre del contador es par. Los contadores siempre se deben inicializar antes de utilizarlos dentro de la estructura repetitiva

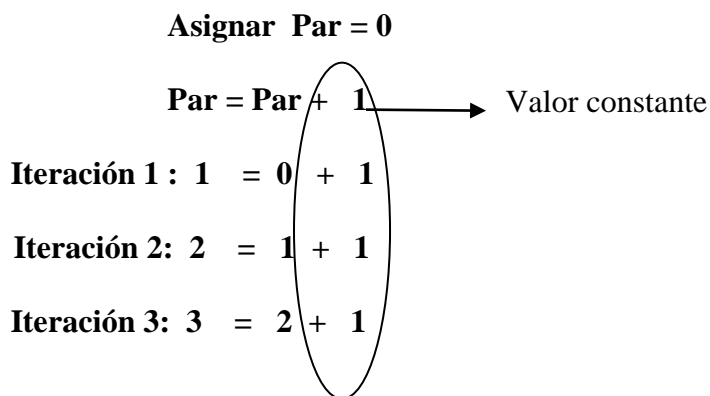
**Asignar Par = 0**

$$\text{Par} = \text{Par} + 1 \longrightarrow \text{Valor constante}$$

**Iteración 1 : 1 = 0 + 1**

**Iteración 2: 2 = 1 + 1**

**Iteración 3: 3 = 2 + 1**



### 4.3.2 Acumulador

Un acumulador o totalizador es una variable cuya misión es almacenar cantidades variables resultantes de suma sucesivas

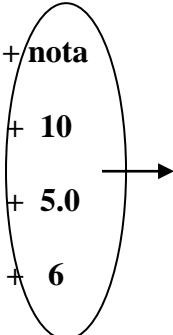
$$\text{Acum} = \text{Acum} + \text{Variable}$$

Al igual que el contador se debe inicializar el acumulador es decir asignarle un valor de inicio y puede ser de cualquier tipo de dato. El acumulador puede ser entero o real y se puede utilizar con cualquier operador aritmético.

Ejemplo: el nombre del acumulador es Media y se inicializa con 0

**Asignar Media = 0**

**Media = Media + nota**

<b>Iteración 1:</b>	<b>10</b>	<b>=</b>	<b>0</b>	<b>+</b>	<b>10</b>	
<b>Iteración 2:</b>	<b>15</b>	<b>=</b>	<b>10</b>	<b>+</b>	<b>5.0</b>	
<b>Iteración 3:</b>	<b>21</b>	<b>=</b>	<b>15</b>	<b>+</b>	<b>6</b>	

Valor variable en cada iteración

### 4.3.3 Estructuras Para (for)

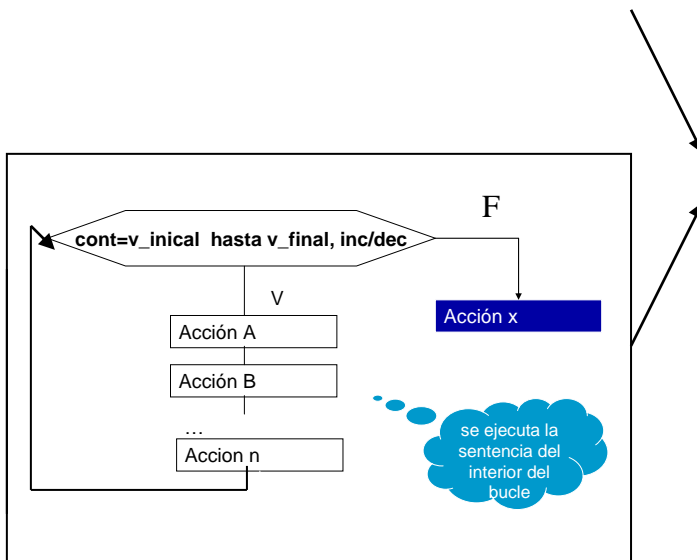
En esta estructura el número de iteraciones es fija. Ejecuta las acciones del cuerpo del bucle un número específico de veces y de modo automático controla el número de iteraciones o pasos a través del cuerpo del bucle.

La sentencia *para* requiere que se conozca por anticipado el número de veces que se ejecutan las sentencias del interior del bucle, se conoce el inicio y el fin del contador que se va incrementado o decrementando en cada iteración hasta llegar al valor final.

### PSEUDOCODIGO

```
Para (cont=V_Inicio hasta V_final, inc/dec)
  Inicio
    Acción A
    Acción B
    .....
    Acción N
  Fin_para
```

```
for (cont=V_I ; cont<=V_F; incr/dec)
{
    C++
    Sentencia B;
    .....
    Sentencia N;
}
```



### Ejemplo: Encontrar el factorial de un numero

#### Pseudocódigo

##### INICIO: FACTORIAL

Declarar fact, num, cont son enteros

Mostrar (“INGRESE EL NUMERO PARA EN CONTRAR SU FACTORIAL”)

Leer (num)

Asignar fact =1

Para (cont=num hasta 1; -1)

Inicio

Asignar fact=fact \* num

Fin\_para

Mostrar(“EL FACTORIAL ES :”, fact)

FIN:FACTORIAL

#### En C++

```
#include <iostream>
using namespace std;
int main( )
{
```

```

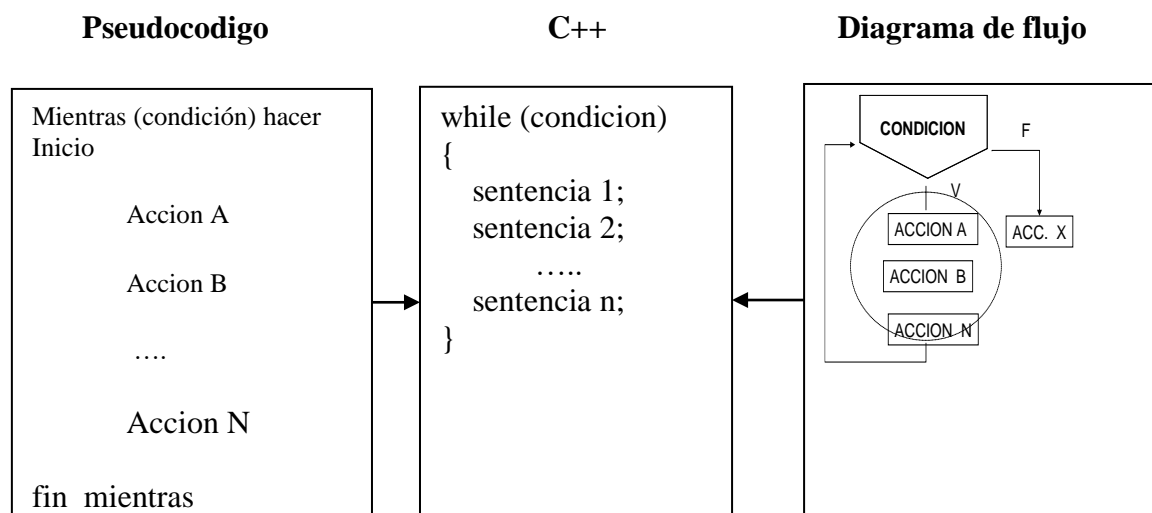
int fact, num, cont ;
cout<<"INGRESE EL NUMERO PARA ENCONTRAR SU FACTORIAL: ";
cin>>num;
fact =1;
for(cont=1;cont<= num; cont=cont+1)
{ fact=fact * cont;
}
cout<<"EL FACTORIAL ES :"<< fact<<endl;
system("PAUSE");
}

```

#### 4.3.4 Estructura Mientras(do while)

La estructura repetitiva mientras es aquella en la que el número de iteraciones no se conoce por anticipado y el cuerpo del bucle se repite *mientras* se cumple una determinada condición. Por esta razón a estos bucles se les denomina bucles condicionales.

Cuando la sentencia mientras se ejecuta, el primer paso es la evaluación de la expresión lógica. Si se evalúa a falso, ninguna acción se realiza y el programa prosigue en la siguiente sentencia después del bucle. Si la expresión lógica se evalúa a verdadera, entonces se ejecuta las sentencias contenidas dentro del cuerpo del bucle y se evalúa de nuevo la expresión. Este proceso se repite *mientras* que la expresión lógica sea verdadera.



#### Ejemplo

Contar los números enteros positivos introducidos por teclado. Se consideran dos variables enteras NUMERO y CONTADOR (contará el número de enteros positivos). Se supone que se leen números positivos y se detiene el bucle cuando se lee un número negativo o cero.

#### PSEUDOCODIGO

#### C++

**INICIO:CONTAR\_POSI**

Declarar numero, contador son enteros

Asignar contador =0

Mostrar("INGRESE UN NUMERO")

Leer (numero)

**Mientras (numero >0 ) hacer**

Inicio

Asignar contador=contador + 1

Mostrar("INGRESE UN NUMERO")

Leer (numero)

**Fin \_mientras**

Mostrar("LA CANTIADAD DE NUMEROS POSITIVOS ES", contador )

**FIN: CONTAR\_POSI**

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

```
void main( )
```

```
{
```

```
int numero, contador;
```

```
contador =0;
```

```
cout<<"INGRESE UN NUMERO";
```

```
cin>>numero;
```

```
while (numero >0 )
```

```
{
```

```
    contador=contador + 1;
```

```
    cout<<"INGRESE UN NUMERO";
```

```
    cin>>numero;
```

```
}
```

```
cout<<"LA CANTIADAD DE NUMEROS POSITIVOS ES"<< contador;
```

```
}
```

**4.3.5 Estructura Repetir (do..... while)**

Las acciones dentro del bucle se repiten un número de veces indeterminado a priori es decir que no se sabe cuantas iteraciones se van a realizar .

Las acciones que forman el cuerpo del bucle se ejecutan por lo menos una vez porque la condición se evalúa al final del bucle por esto es denominado también bucle post-prueba (se evalúa la condición después de ejecutar sus sentencias).

**PSEUDOCODIGO****C++****DIAGRAMA DE FLUJO**

Repetir

Acción A

Acción B

.....

Acción N

Hasta que (condición)

Do

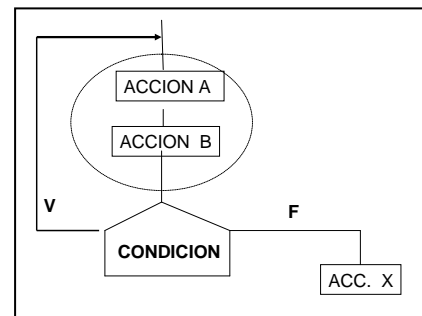
{

sentencia A;

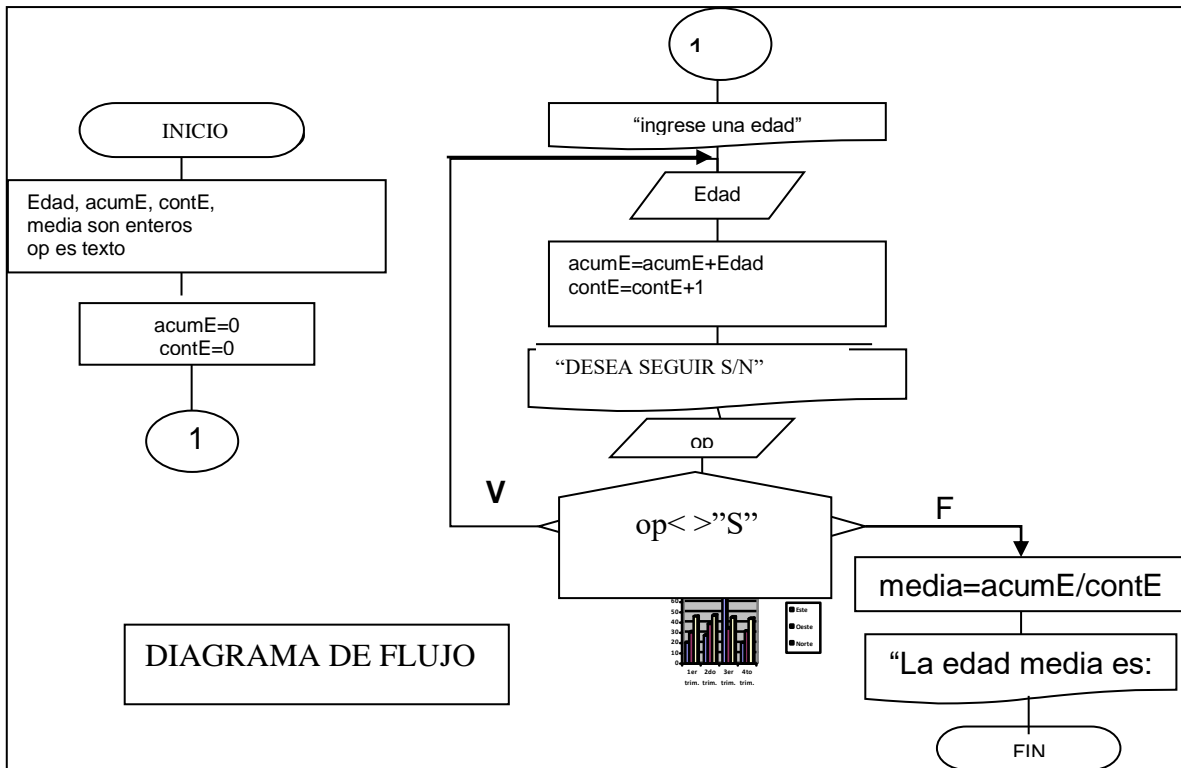
sentencia B;

.....

}while (condicion);



**Ejemplo:** Realizar un algoritmo para encontrar la edad media de n edades hasta que el usuario desee ya no ingresar mas edades



**INICIO:MediaEdad**  
 Declarar edad, acumE, contE, media son enteros  
 Declarar Op es texto  
 Asignar acumE = 0  
 Asignar contE = 0  
 Repetir  
 Inicio  
 Mostrar("ingrese una edad")  
 Leer(Edad)  
 Asignar acumE=acumE+ edad  
 Asignar contE=contE + 1  
 Mostrar("Desea seguir ingresando mas edades S/N")  
 Leer(op)  
 Fin\_repetir  
 Mientras (op ="S")  
 Asignar media=acumE/contE  
 Mostrar ("La edad media es :", media)  
 Fin:MediaEdad

```

#include<iostream.h>
void main()
{
  int edad, acumE, contE, media ;
  char op;
  acumE = 0
  contE = 0
  do
  { cout<<"ingrese una edad";
    cin>>Edad;
    acumE=acumE+ Edad;
    contE=contE + 1;
    cout<<"Desea seguir ingresando mas edades < S/N>";
    cin>>op;
  }
  while (op == 's');
  media=acumE/contE;
  cout<<"La edad media es : "<<media
}
  
```

#### 4.3.6 DIFERENCIAS ENTRE ESTRUCTURAS DE CONTROL REPETITIVAS

- La diferencia entre ambas es que la condición se sitúa al principio (Mientras) o al final (Repetir) de la secuencia de instrucciones. Entonces, en el primero, el bucle continúa mientras la condición es verdadera (la cual se comprueba antes de ejecutar la acción) y en el segundo, el bucle continúa hasta que la condición se hace verdadera (la condición se comprueba después de ejecutar la acción, es decir, se ejecutará al menos una vez).



- La estructura Desde/Para suele utilizarse cuando se conoce con anterioridad el número de veces que se ejecutará la acción y se le conoce como Estructura Repetitiva en lugar de iterativa, para diferenciarla de las dos anteriores.
- Las estructuras Mientras y Para/Desde suelen en ciertos casos, no realizar ninguna iteración en el bucle, mientras que Repetir ejecutará el bucle al menos una vez.

```
#include <iostream>
using namespace std;
int main( )
{
    int fact, num, cont ;
    cout<<"INGRESE EL NUMERO PARA ENCONTRAR SU FACTORIAL: ";
    cin>>num;
    fact =1;
    /* con for */
    for(cont=1;cont<= num; cont=cont+1)
    {
        fact=fact * cont;
    }
    cout<<"EL FACTORIAL ES :"<< fact<<endl;
    system("PAUSE");

    /* con while */
    fact=1;
    cont=1;
    while (cont<=num)
    {
        fact=fact*cont;
        cont=cont+1;
    }
    cout<<"EL FACTORIAL ES :"<< fact<<endl;
    system("PAUSE");

    /* do ... while */
    fact=1;
    cont=1;
    do
    {
        fact=fact*cont;
        cont=cont+1;
    }
    while (cont<=num);
    cout<<"EL FACTORIAL ES :"<< fact<<endl;
    system("PAUSE");

}
```

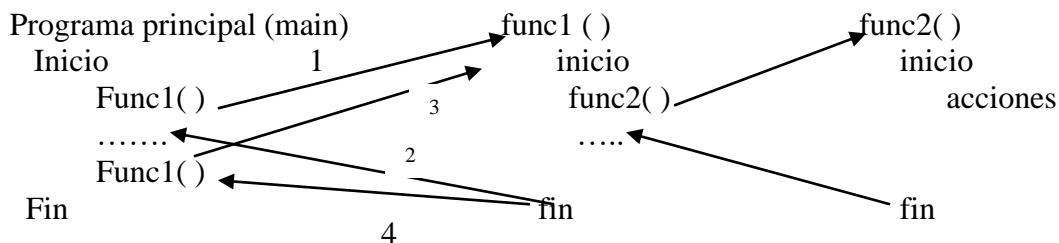
## PROGRAMACION MODULAR

### 4.1 INTRODUCCION

Uno de los métodos más conocidos para resolver un problema es dividir en problemas más pequeños, llamados subproblemas. De esta manera, en lugar de resolver una tarea compleja y tediosa, resolvemos otras más sencillas y a partir de ellas llegamos a la solución a esta técnica se le suele llamar diseño descendente, metodología del divide y vencerás o programación top-down.

A estos subprogramas se les suele llamar módulos, de ahí viene el nombre de programación modular.

En C++ consta al menos de una función **main** y otras funciones alternativas del programa. La ejecución de un programa comienza por la función main, cuando se llama a una función, el control se pasa a la misma para su ejecución y cuando finaliza el control es devuelto de nuevo al modulo que llamo para continuar con la ejecución del mismo a partir de la sentencia que efectuó la llamada. De la siguiente manera se realiza las llamadas.



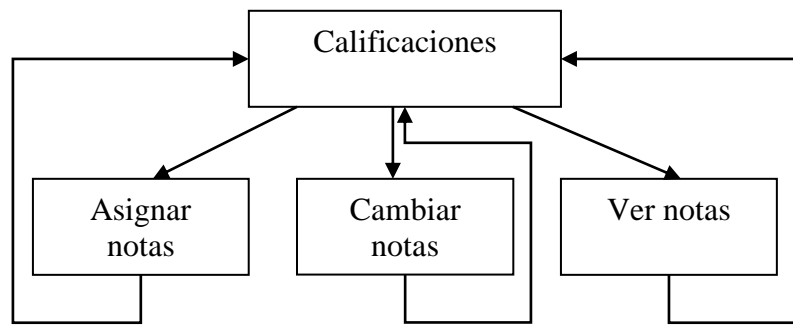
### 4.2 CONCEPTO DE MODULO

Un modulo representa siempre una función o tarea determinada y en general consta de: Un conjunto de instrucciones que se procesan de una sola vez y se referirán mediante un nombre por el que posteriormente serán llamados o invocados desde diferentes puntos de un programa.

Los módulos deben ser pequeños para que sean claros y de poca complejidad, el tamaño máximo de un modulo sea correspondiente a una pagina de impresión. Un modulo debe detener un punto de entrada y un punto de salida y se clasifican:

- Función
- Procedimiento

Por ejemplo un profesor quiere crear un programa para gestionar las notas de sus alumnos. Quiere que dicho programa le permita realizar tareas tales como asignar notas, cambiar notas, ver las notas según código de estudiante, etc.



### 4.3 PARAMETROS

Los parámetros también se les conoce como argumentos y tienen la misión de comunicar al procedimiento o función con el programa que lo llama.

Al llamar a un subprograma los datos de entrada se le envían a través de los parámetros de entrada y de igual forma al terminar el programa los resultados o datos de salida se envían a través de parámetros de salida.

Existen dos métodos de paso de parámetros :

- por valor
- por referencia

#### 4.3.1 Paso de parámetros por valor

Un parámetro por valor es un dato que tiene un valor determinado que se pasa al correspondiente parámetro formal. Los parámetros por valor son parámetros de entrada y por consiguiente no pueden ser modificados sus valores por el subprograma por lo tanto entran con un valor y salen con el mismo valor .

#### 4.3.2 Paso de parámetros por referencia

Es aquel en que pasa la variable y no el valor, lo que permite pueda ser modificado por el subprograma llamado, es decir que entran con un valor y salen del modulo con un valor cambiado , se los conoce de entrada y salida.

### 4.4 PROCEDIMIENTO

Un procedimiento es un subprograma que realiza una tarea específica que proporciona cero, uno o varios valores en función de los parámetros definidos en su formato.

Para invocarlo, es decir, para hacer que se ejecute, basta con escribir su nombre en el cuerpo de otro modulo o en el programa principal.

## PSEDOCODIGO

## C++

Procedimiento nombre (p1 es tipo, p2 es tipo, ...)  Inicio  Acciones  Fin_procedimiento	<pre>void nombre_proc (tipo p1, tipo p2,...) {     Sentencias ; }</pre>
---	---

## Llamada a un procedimiento

Pseudocodigo  
Llamar Nombre\_Procedimiento (p1,p2, .....)

C++  
Nombre\_procedimiento (p1, p2.....);

Ejemplo : Desarrollar un programa para encontrar el área de un triangulo utilizar un procedimiento para calcular el área de un triangulo

### INICIO:AREA\_ TRIANGULO

**Procedimiento Area** ( base es real , altura es real )

Inicio

    Declarar R\_area es real

    Asignar  $R\_area = (base * altura) / 2$

    Mostrar (“ el area es : ”, R\_area )

Fin\_Area

### Programa principal

Inicio

    Declarar b, a son reales

    Mostrar (“ ingrese la base y la altura ”)

    Leer ( b, a)

    Llamar **Area**(b, a)

Fin

FIN: AREA\_ TRIANGULO

**En C++**

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

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

```
void Area ( float base , float real )  
{  
    float R_area;  
    R_area = (base * altura )/ 2;  
    cout<<" el area es : "<<R_area;  
}
```

```
void main ( )  
{ float b, a ;  
    cout<<" ingrese la base y la altura ";  
    cin>>b>> a;  
    Area(b, a);  
}
```

## 4.5 FUNCION

Una función es un subprograma que proporciona o retorna un valor o resultado según sea sus argumentos (parámetros de entrada).

Las funciones pueden ser :

- Externas
- Internas.

4.5.1 **Funciones internas** : Las funciones internas son definidas por el programa  
Algunas funciones matemáticas mas conocidas en C++ son :

- cos(x): coseno
- sin(x ): seno
- abs(x) : valor absoluto
- pow(x,y): eleva cualquier numero a cualquier exponente
- sqrt(x): raíz cuadrada /

La librería o directriz que se debe utilizar es <math.h>

4.5.2 **Funciones externas**: Son desarrolladas por el usuario. Una función se invoca cuando se le hace regencia, mediante su nombre y lista de parámetros actuales en cualquier instrucción donde se pueda usar una constante o variable.

## Pseudocódigo

```
Función nombre_fun (p1 es tipo, ..) es tipo
Inicio
    Acciones
    Retornar variable / valor
Fin_funcion
```

## C++

```
Tipo nombre_fun(tipo p1...)
{
    Sentencias ;
    return variable/ valor ;
}
```

Para llamar a una función en pseudocódigo o en C++ se le asigna el valor que va a retornar de la función a una variable del mismo tipo de la función

Variable= nombre\_funcion( p1, p2 ....)

Ejemplo: realizar un algoritmo para determinar el promedio de 3 notas

INICIO :PROMEDIO

**Función** promedio ( ) es real

Inicio

Declarar n1,n2,n3, Prom son reales

Escribir (“ingrese tres notas “)

Leer ( n1,n2 , n3 )

Asignar Prom = (n1+n2+n3)/3

Retornar Prom

Fin\_promedio

**Programa principal**

**Inicio**

Declarar P es real ;

Asignar P= promedio( )

Mostrar ( “ el promedio es :” , P)

**Fin\_algoritmo**

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

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

```
float promedio ( )
```

```
{
```

```
    float n1,n2,n3, Prom;
```

```
    cout<<“ingrese tres notas”;
```

```
    cin>> n1;
```

```
    cin>>n2;
```

```
    cin>> n3;
```

```
    Prom = (n1+n2+n3)/3 ;
```

```
    return Prom;
```

```
}
```

```
void main ( )
```

```
{
```

```
    float P ;
```

```
    P= promedio( ) ;
```

```
    cout<< “ el promedio es :” << P;
```

```
}
```

## 4.6 AMBITO DE VARIABLES

### 4.6.1 Variables locales

Es una variable que está declarada dentro de un subprograma, y se dice que es local al subprograma. Y lo que la caracteriza es que su valor sólo está disponible mientras se

ejecuta el subprograma. Dicho de otra manera, el programa principal no tiene conocimiento alguno de las variables locales de sus procedimientos y funciones.

#### **4.6.2 Variables globales**

Las variables globales están declaradas en el programa principal y su valor está disponible tanto en el cuerpo del programa principal como en el de cualquiera de los subprogramas declarados.

### ARREGLOS

#### 6.1 INTRODUCCION

Un array o arreglo es un conjunto de datos que se almacenan en memoria de manera seguida con el mismo nombre. Es una colección de datos del mismo tipo, cada una de ellas se llama elemento y posee una posición dentro del arreglo llamado índice que se incrementa de uno en uno.

Se clasifican:

- Unidimensionales
- Bidimensionales

#### 6.2. Arreglos Unidimensionales

Un arreglo unidimensional llamado también vector maneja un solo índice que indica la posición de un elemento del vector. La primera posición del array es la posición 0.

Podríamos agrupar en un array una serie de elementos de tipo enteros, flotantes, caracteres, objetos, etc.

Crear un vector en C++ es sencillo, seguimos la siguiente sintaxis:

Tipo nombre[tamaño\_Max];

Ejemplo: int A[30];

Ejemplo 1 : Realizar un algoritmo para almacenar números enteros y mostrar solo los números pares del vector.

Inicio: Pares

Declarar v[50], tam, ind son enteros

escribir("Ingrese la cantidad de números a almacenar en el vector")

leer(tam)

Para ind=0 hasta tam-1, +1

    mostrar("ingrese un numero ")

    leer(v[ind])

fin\_para

Para ind=0 hasta tam-1, +1

    Si v[ind] mod 2= 0 entonces

        mostrar(v[ind])

fin\_si



fin\_para  
fin: Pares

Ejemplo 2 : Realizar algoritmo y un programa en C++ para sumar 2 vectores a y b y poner el resultado en un tercer vector c:

Pseudocodigo

Inicio

Procedimiento sumar(A[ ]: entero; B[ ]: entero; c[ ]: entero; tam : entero)

Inicio

Declarar ind es entero

Para ind=0 hasta tam-1, +1

C[ind]=A[ind] + B[ind]

fin\_para

fin\_procedimiento

Procedimiento imprimir( V[ ]: entero; tam : entero)

Inicio

para ind=0 hasta tam-1, +1

Escribir (V[ind], "-")

fin\_para

fin\_procedimiento

Programa principal

Inicio

Declarar tam: entero

escribir("Ingrese la cantidad de numeros a almacenar en el vector")

leer(tam)

Declarar A[20]:entero

Declarar B[20]:entero

Declarar C[20]:entero

para ind=0 hasta tam-1, +1

Inicio

leer(A[ind])

leer(B[ind])

fin\_para

escribir ("vector A")

imprimir (A, tam)

escribir ("vector B")

imprimir (B ,tam)

suma(A,B,C, tam)

escribir (" Suma de vectores")

imprimir (C,tam)

Fin

## El algoritmo desarrollado en c++

```
#include <iostream>
using namespace std;
void sumar(int a[], int b[], int c[],int tam) {
    for (int i = 0; i < tam; i++) {
        c[i] = a[i] + b[i];
    }
}

void imprimir(int v[], int tam)
{
    for(int i = 0; i < tam; i++)
    {
        cout << v[i] << endl;
    }
    cout << endl << endl;
}

int main()
{
    int tam;
    cout << "Ingresa la cantidad de numeros a almacenar en el vector" <<
endl;
    cin >> tam;

    int a[20];
    int b[20];
    int c[20];

    for(int i = 0; i < tam; i++) {
        cout<<"Escriba elemento del vector A:";
        cin>>a[i];
        cout<<"Escriba elemento del vector B:";
        cin>>b[i];
    }
    cout << "Vector A " << endl;
    imprimir(a, tam);

    cout << "Vector B " << endl;
    imprimir(b, tam);

    sumar(a, b, c, tam);
    cout << " Suma de Vectores A y B " << endl;

    imprimir(c, tam);
}
```

Entonces para tomar en cuenta:

- Todo vector debe tener definido un tipo de dato.
- Todo vector necesita de una dimensión o tamaño

## 6.3. Arreglos Bidimensionales

Una matriz es un vector de vectores o un también llamado array bidimensional. La manera de declarar una matriz en C++ es similar a un vector:

tipo nombre\_Matriz[Max\_filas][Max\_cols];

Ejemplo: Declarar Mat[15][25]:entero -----> Pseudocodigo

int Mat[15][25]; -----> c++

Las matrices también pueden ser de distintos tipos de datos como char, float, double, etc. Las matrices en C++ se almacenan al igual que los vectores en posiciones consecutivas de memoria.

La forma de acceder a los elementos de la matriz es utilizando su nombre e indicando los 2 subíndices que van en los corchetes.

Ejemplo 1 : Llenar una matriz con números enteros.

Para i=0 hasta tam\_filas, +1  
Para j=0 hasta tam\_cols, +1

Leer( Mat[ i ][ j])

En C++

```
1 for(int i = 0; i < tam_filas; i++) {  
2   for(int j = 0; j < tam_cols; j++) {  
3     cin>>Mat[i][j] ;  
4   }  
5 }
```

Ejemplo 2: Mostrar los elementos de la matriz

Para i=0 hasta tam\_filas, +1

Para j =0 hasta tam\_Cols, +1

Escribir ( Mat[i][j])

En C++

```
1 for(int i = 0; i < tam_filas; i++) {  
2   for(int j = 0; j < tam_cols; j++) {
```

```

3     cout<<Mat[i][j] ;
4 }
5 }

```

### Ejercicios propuestos

1. Llenar un vector con las ventas que se realizaron durante una semana. Mostrar la venta mayor y el día. Considerar que puede haber varios días con la venta mayor.
2. Llenar un vector con n notas y ordenar en forma ascendente, luego insertar una nota en la posición que le corresponde.
3. Visualizar las siguientes matrices : La dimensión de la matriz en n \* m

1 2 3 4	1	1 3 5
2 4 6 8	1 2	2 4 6
3 6 9 12	1 2 3	7 9 11
4 8 12 16	1 2 3 4	8 10 12

4. Realizar la multiplicación de 2 matrices