

### **Compilación**

En programación entendemos compilación como el proceso de convertir unas instrucciones hechas en un lenguaje de programación a un lenguaje de máquina. Cuando compilamos un programa, el PC convierte todas las instrucciones que le dijimos y nos da como resultado la ejecución de todas las instrucciones que se le hayan programado.

### **Variables Acumuladoras**

Es una variable que suma sobre sí misma un conjunto de valores, para de esta manera tener la suma de todos ellos en una sola variable. Por ejemplo la suma de los 10 primeros números naturales en una variable llamada "Sumatoria", lo que hace es sumar de la siguiente manera:  $(1+2+3+4+5+6+7+8+9+10)$ .

### **Estructuras Repetitivas**

Las estructuras repetitivas son aquellas que sirven para evitar la molestia de andar repitiendo una acción varias veces. Estas sirven para que una acción se ejecute un determinado número de veces, y depende del cumplimiento de una determinada condición.

**FOR (Para):** estructura repetitiva con la cual podemos ejecutar unas instrucciones un número finito de veces, a diferencia del while que se repite una cantidad de veces según la condición se cumpla, el for solo se le da el número de veces que se quiere repetir.

### **VARIABLES**

Una variable es un espacio en memoria que se reserva para la ejecución de programas. Le corresponde un tipo de dato como números (enteros o flotantes); carácter ('T';'g';'Q';'l';'a';'B'; cadenas de caracteres como 'Luciano'; 'situación'; -> muchos lenguajes de programación distinguen mayúsculas y minúsculas) booleano (Verdadero o Falso). Muchas veces a estos datos (salvo a la cadena de caracteres) se los conoce como dato primitivo. Estos datos se guardan en porciones de memoria para luego ser utilizados.

Varios lenguajes de programación tienen un tipado, es decir que hay que especificar o referir a cómo vamos a declarar los tipos de variables: enteras, cadenas, caracteres, etc. Mientras que en tipado débil no se indica el tipo de variable y se puede cambiar sobre la marcha.

Además, los códigos escritos sea el tipado que sea es menester que estén bien indentados (tabulados) y comentados para saber qué hace específicamente cada parte o bloque del código. Esto se hace para tener referencias a futuro a la hora de actualizar el programa ya escrito.

Las variables toman los últimos valores asignados al principio o mutan de acuerdo al ingreso y/o modificación de la información.

### **Ciclos**

Un ciclo es una repetición de una instrucción hasta que deje de repetirse su condición. En programación un ciclo se usa cuando queremos repetir un conjunto de instrucciones un número finito de veces. (Sí intentamos que un ciclo sea infinito entra en un loop o bucle del cual no puede salir, en computadores de hogar esto puede llevar a la destrucción de algunos componentes si no se reinicia a tiempo). Para que este ciclo sea finito necesitamos de una

condición, si esta deja de cumplirse el ciclo se da por finalizado o seguirá repitiéndose hasta cumplirla.

### **¿Qué es la estructura secuencial?**

La estructura secuencial, realiza una acción o tarea y luego realiza la siguiente acción, en orden. Una secuencia puede contener cualquier cantidad de tareas, pero no ramificarse u omitir ninguna de las tareas. Una vez comience con una serie de acciones en secuencia, debe continuar paso a paso hasta que finalice la secuencia.

#### **Asignación**

Simples:  $(a = 15)$

Contador:  $(a = a+1)$

Acumulador:  $(a = a+b)$

De trabajo:  $(a = c+b*2/4)$

#### **Lectura y Escritura**

Leer a,b;

Escribir "La cantidad total es: ", R;

Definir xuno, xdos, auxiliar como entero;

//Lectura

leer xuno;

leer xdos;

//Asignación

Auxiliar <- xuno;

Xuno <- xdos;

Xdos <- auxiliar;

//Escritura

Escribir xuno;

Escribir xdos;

## Estructuras condicionales

Son estructuras que nos permiten controlar el flujo de un programa mediante una condición que nosotros establecemos. Dicha condición debe ser booleana y debe brindar como resultado un valor Verdadero o Falso.

**\*\***(con flujo nos referimos a dónde se va dirigir nuestro programa)

### **Si condición Entonces**

Instrucciones

### **FinSi**

Las palabras en negrita son reservadas al sistema, en inglés se las conoce como **If, Then, Else**

Sí es Verdadero o true la condición, se proceden a ejecutar la o las instrucciones recorriendo el ciclo. De ser Falso o false no se recorre el ciclo y se pasa por alto.

**\*\***La condición es establecida por el programador y es la que va a ser analizada.

**\*\***El condicional compuesto puede tener un resultado distinto o tener un resultado sí la condición da por resultado falso. Sus palabras reservadas son: **sí, entonces, sino, finsi**. (FinSi se usa en PsEint, en C++ el condicional se cierra con la llave})

## Operador Mod

Mod es el resto de la división entera, es decir, el residuo que sobra al calcularse entre dos números que no tienen decimales.

Usos comunes: par o impar de un número, averiguar si un número es múltiplo de otro. Devolución del resto de la división.

## Sub-Programa

Son un conjunto de instrucciones o bloque de código que realizan una labor específica. Nacen de la necesidad de no repetir innecesariamente un trabajo ya hecho. Pueden invocarse desde el cuerpo del programa principal cuantas veces se desee. Facilitan la descomposición de un problema grande en partes más pequeñas abordables de manera más sencilla.

**\*{variable\_de\_retorno}**: si su función/subproceso calcula y devuelve algún valor, complete aquí el nombre de la variable que utilizará para almacenar ese valor; si su función/subproceso no devuelve nada puede eliminar la variable y la flecha de asignación.

**\*{nombre}**: introduzca aquí el nombre que recibirá la función/subproceso.

**\*{argumentos}**: introduzca aquí la lista de argumentos que requiere su función, utilizando comas para separarlos, e ingresando el identificador para cada uno y opcionalmente las palabras clave "por valor" o "por referencia" para indicar el tipo de pasaje; si la función/subproceso no requiere argumentos puede dejar este campo en blanco, y opcionalmente omitir los paréntesis.

FUNCION Variable de retorno <- nombre (argumentos)

**\*\* Aclaraciones generales \*\***

Antes de llamar a un subprograma hay que declararlo. Todos los SP tienen un nombre, el cual indica sin ambigüedad qué hace el subprograma; a la hora de nombrarlos se recomienda que tenga un verbo ya que por lo general realizan una acción, ejemplo de esto es: “subprograma(calcularEnteros)”. No se usan siglas o letras al azar ya que es necesario identificar lo realizado con una simple mirada.

Cuentan con un listado de parámetros los cuales tienen un nombre y un tipo, los cuales interactúan con el código del SP a través de ese nombre y ese tipo.

Luego encontramos el cuerpo del SubPrograma, que puede contener una línea de código o varias de ellas. En cuanto a esto último lo recomendable es que sean unas pocas líneas ya que debe resolver un problema específico. Generalmente todo lo que se puede colocar en un programa principal se puede colocar en un SP. Los subprogramas que no devuelven un valor se denominan **procedimientos** ya que solo se encargan de realizar las instrucciones.

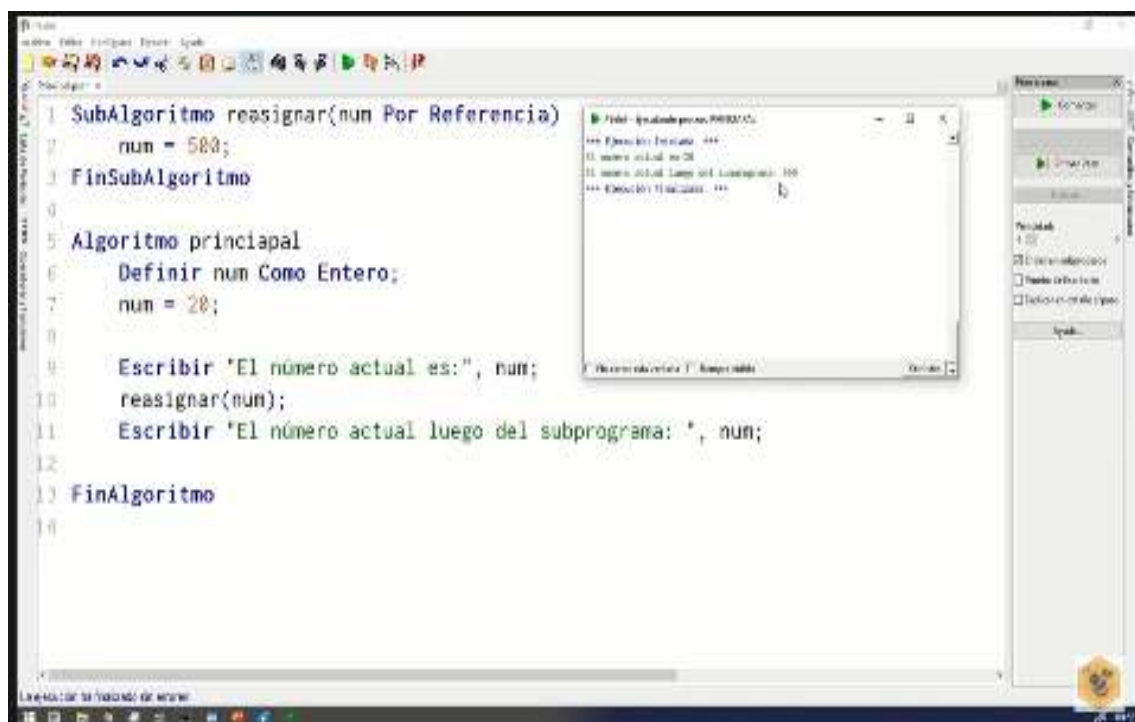
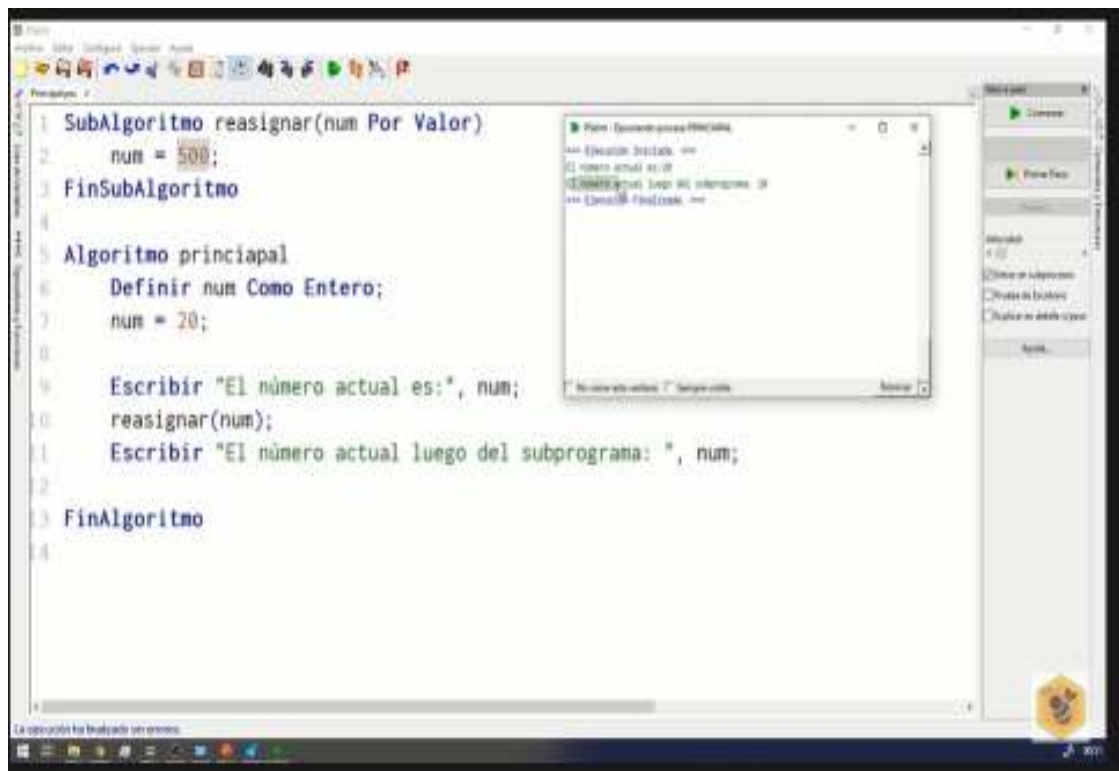
Otros si retornan un valor al programa principal, el tipo de dato retornado se aclara en la cabecera del SP, y el valor que queremos devolver debe ser indicado en la última línea del cuerpo y debe coincidir con lo declarado en la cabecera... Estos SP se denominan **funciones**.

**Paso de parámetros**

Es la forma en la que se transfieren los argumentos de una invocación a los parámetros formales de un subprograma. Existen dos formas: por valor y por referencia.

Por VALOR: se crea una copia del valor para ser usada de manera local, de forma que los cambios que se haga sobre esta sólo permanecen dentro del subprograma.

Por REFERENCIA: paso de una referencia a la dirección de memoria donde se encuentra almacenada una variable y los cambios permanecen dentro y fuera del subprograma.



## ¿Qué imprime el siguiente programa?

```
SubAlgoritmo aumentarSueldo(sueldoRef Por Referencia)
    sueldoRef = sueldoRef + (sueldoRef * 0.1);
FinSubAlgoritmo

Algoritmo pasoPorReferencia

    Definir sueldo Como Entero;
    sueldo = 35600;

    aumentarSueldo(sueldo);

    Escribir "Su sueldo es: $", sueldo;

FinAlgoritmo
```

Annotations:

- sueldoRef** apunta a la misma dirección de memoria que **sueldo**
- sueldoRef** y **sueldo** valen 39160
- Defino la variable **sueldo** de tipo Entero
- sueldo** vale 35600
- Invoco **aumentarSueldo** y le paso **sueldo** como argumento
- Imprime:** Su sueldo es \$39160

\*\*-----\*\*

Estos se definen fuera del algoritmo principal, una vez definidos uno o más subprogramas, cualquiera de ellos puede ser invocado en cualquier parte del programa principal. Pueden o no recibir parámetros. Estos indican las variables que van a recibir.

Las variables de los SP no se interfieren entre sí, ni se puede usar una variable del SP 1 en el SP2. Las variables son locales al sub-programa, por lo cual podemos tener la variable "numero" en 2 SP pero estas solo funcionan de forma local al método.

### Tipos y su declaración

- 1) Se declara primero la palabra clave **Función** (hablando de **PsEint**)
- 2) Variable que retorna.
- 3) Nombre del método.
- 4) Parámetros (puede tener 1 o más, o puede tener parámetros vacíos).

Las funciones retornan un valor (variable) y siempre se retorna al final del bloque de código. Este valor siempre se declara antes del nombre de la función.

### Procedimiento (En **PsInt**)

- 1) Se declara primero la palabra clave **SubAlgoritmo**.
- 2) Nombre del método.
- 3) Parámetros (puede tener 1 o más, o puede tener parámetros vacíos).

### Pasaje de Variables por valor

Desde el algoritmo principal mandamos el valor de una variable como parámetro a un subprograma.

El SP recibe la copia de ella, y cualquier modificación que haga no se verá reflejada en el algoritmo principal ni en otra función o procedimiento, pero sí dentro del Subprograma.

Es una forma de enviar parámetros por defecto.

### Pasaje de variables por referencia

En lugar de pasar el valor alojado en una variable se pasa la referencia a esa variable. Dicha referencia de una variable es la dirección de memoria en donde está alojada. En estos casos, los cambios que realice el subprograma se verán reflejados fuera de este.

### \*\*\*Procedimientos

Son los métodos que no tienen retorno de datos. Ejecutan un bloque de instrucciones para realizar una acción particular sin devolver un resultado

### \*\*\*¿Para qué se usan?

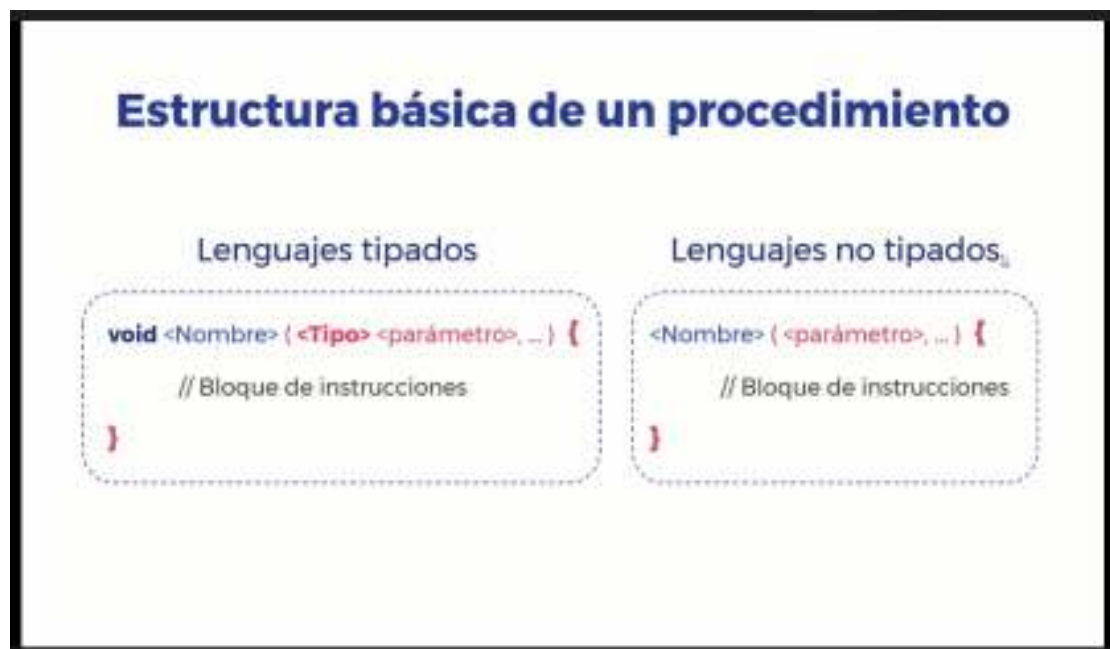
Para imprimir mensajes, cambiar el valor de una o varias variables ya existentes, controlar datos.

### \*\*\*Funciones

Son los métodos que retornan un valor de un tipo definido y el valor retornado debe llegar a una variable del mismo tipo del retorno. Es decir que al momento de invocar una función se debe hacer inicializando o dando valor a una variable del tipo de dato del retorno.

### \*\*\*¿Para qué se usan?

Para realizar operaciones, asignarle un nuevo valor a una variable.



## Estructura básica de una función

### Lenguajes tipados

```
<Tipo> <Nombre> ( <Tipo> <parámetro>, ... ) {  
    return <Dato de retorno>  
}
```

### Lenguajes no tipados

```
<Nombre> ( <parámetro>, ... ) {  
    return <Dato de retorno>  
}
```

### \*\*\* Vector, array o arreglo

¿Qué es un Array? Es una colección de datos de tipos similares. Es un contenedor de datos de un solo tipo. Por ejemplo, se puede crear un array que pueda contener 100 valores de tipo entero. Nos permiten guardar grandes cantidades de información, solo pueden almacenar un tipo de dato, tengo que definir qué tipo de datos van a ser, el tamaño es fijo ya que nosotros lo definimos con una capacidad cuando la creamos.

\*\*Definimos el tipo de dato a guardar.

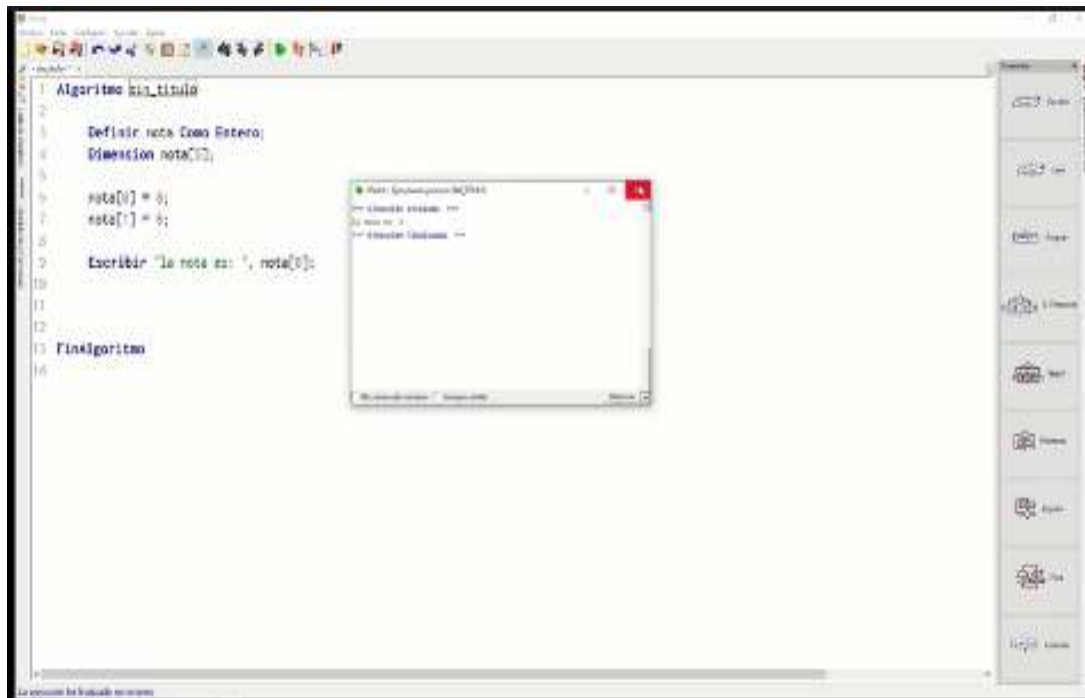
\*\*Definimos el tamaño de la dimensión.

\*\*Podemos llenar de manera manual los subíndices, pero pueden ser dinámicos.

\*\*Usamos un bucle “para” para llenar cantidades de datos grandes y dinámicos.

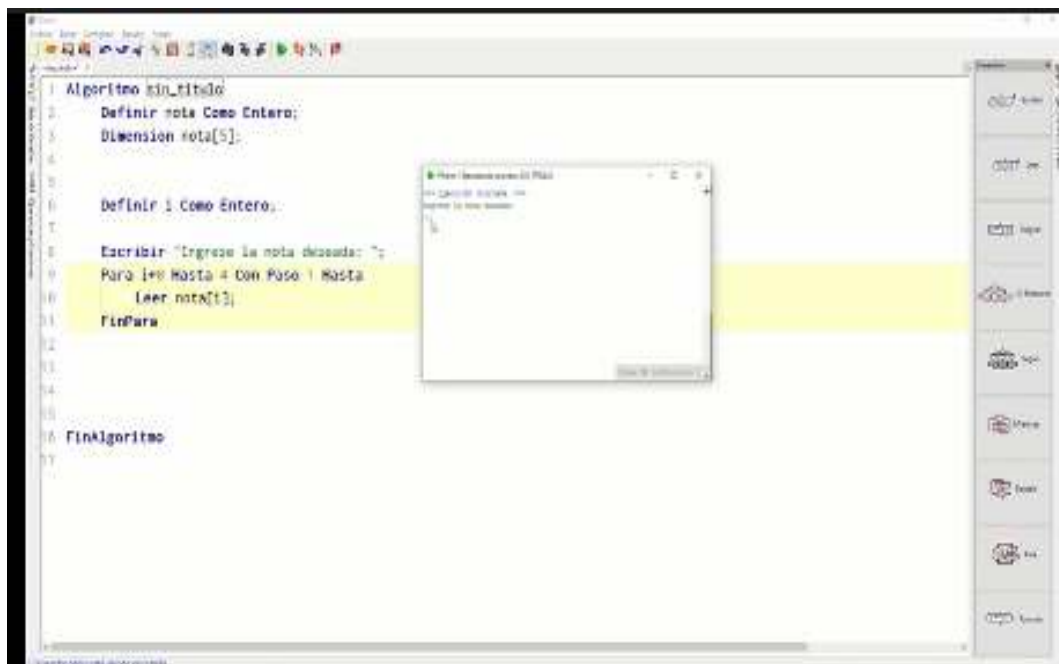
\*\*\*Ejemplo llenado manualmente.

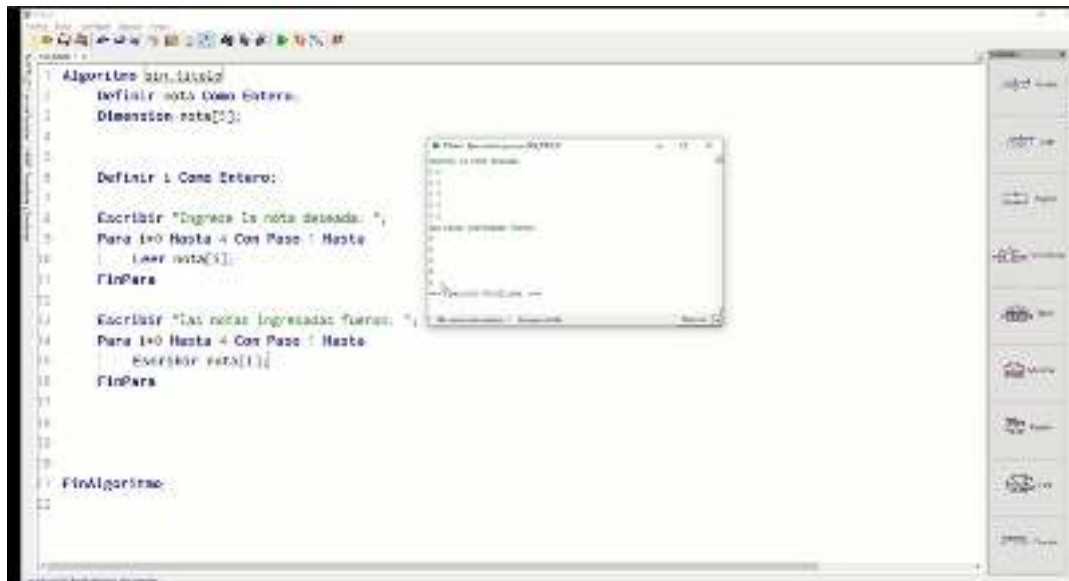




### Ejemplo de Bucle para en funcionamiento

Utilizamos un bucle “Para” para el llenado del arreglo y un segundo bucle para mostrarlo.





## MATRICES Y VECTORES

Ambas son estructuras de datos estáticas, o sea que separan memoria física para almacenar datos y que no puede ser modificada en tiempo de ejecución. La diferencia está en la manera de acceder a dichas estructuras; mientras que los **VECTORES** son accedidos con un subíndice, las **MATRICES** usan dos subíndices. En el momento de crear una MATRIZ, la memoria central separa espacio en posiciones consecutivas que gráficamente representan casilla de filas y columnas.

### **\*\*Arreglos Unidimensionales**

#### **Otra definición:**

Un arreglo o *array* es una estructura que permite almacenar una colección finita y ordenada de elementos del mismo tipo bajo un identificador (nombre). El tamaño o *dimensión* se declara en un primer momento y puede ser de una o más dimensiones. Esta estructura está compuesta por dos partes: los componentes y los índices.

El componente son los valores almacenados (números o caracteres).

Los índices son la posición de dichos valores, comenzando en la posición cero (0) hasta N.

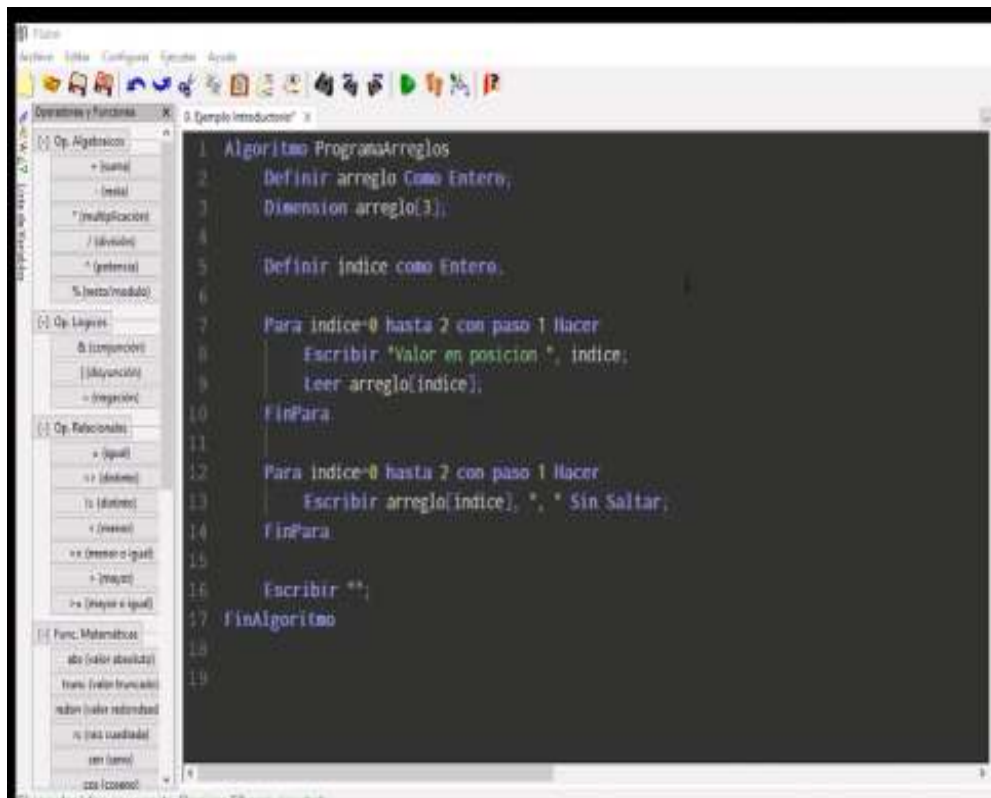
Son arreglos con una sola dimensión.

## Arreglos Unidimensionales

Son arreglos con una sola dimensión.

Ejemplo de una declaración de un arreglo en Pseint:

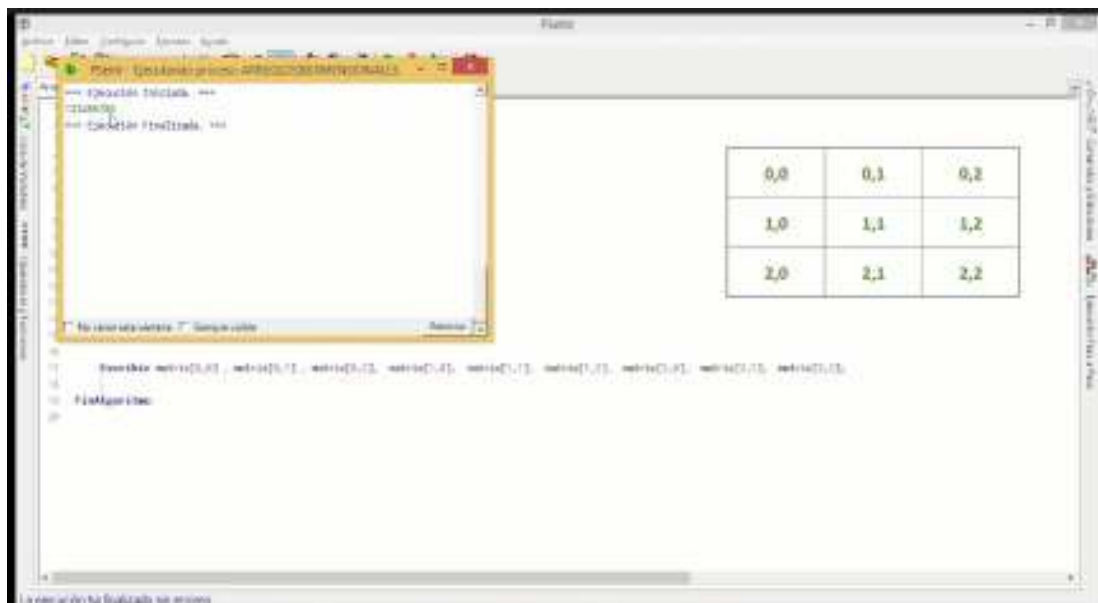
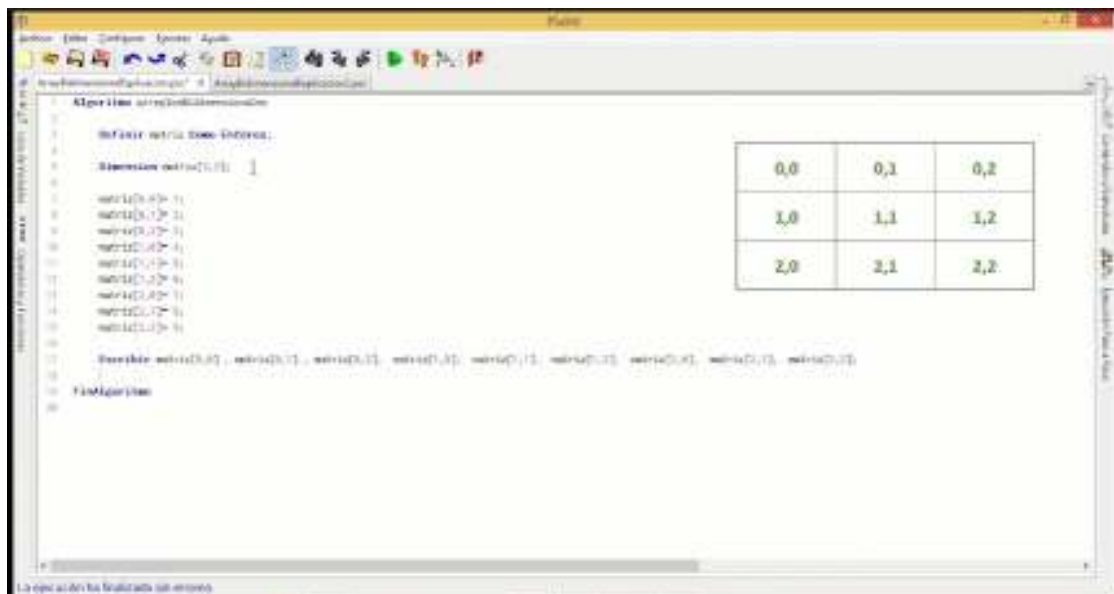
```
Definir arreglo Como Caracter;  
Dimension arreglo[5];  
  
Para indice=0 hasta 4 con paso 1 Hacer  
    Escribir "Valor en posicion ", indice;  
    Leer arreglo[indice];  
FinPara
```



## Arreglo BiDimensional

En estos arreglos cada elemento tiene una posición que se identifica mediante dos índices: el de su fila y el de su columna. Se conocen como tablas de valores. Cada elemento está simultáneamente en una fila y en una columna.

En matemáticas, a los arreglos bidimensionales se les llama matrices y son muy utilizados en problemas de ingeniería.



## RECURSIVIDAD

### \*\*\*Recursividad

Es una técnica utilizada en programación que nos permite llamar al subprograma dentro del mismo. De allí viene su nombre y gracias a esto podemos utilizar recursividad en lugar de bucles para resolver determinados tipos de problemas. Es una función que se invoca a si misma con una condición de terminación.

Cuando se usa:

- \*) Cuando la solución al problema es más simple si se lo divide en versiones más pequeñas de sí mismo.
- \*) al trabajar con estructuras de datos de naturaleza recursiva (listas, árboles).
- \*) cuando la forma iterativa es más compleja y produce código más difícil de comprender.

\*) en lenguajes o paradigmas de programación orientadas a la recursividad.

## RECURSIVIDAD

La recursividad (recursión) es aquella propiedad que posee una función por la cual puede llamarse a sí misma. Aunque la recursividad se puede utilizar como una alternativa a la iteración, una solución recursiva es normalmente menos eficiente, en términos de tiempo de computadora, que una solución iterativa, debido a las operaciones auxiliares que llevan consigo las invocaciones suplementarias a las funciones.

Sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación. Diversas técnicas algorítmicas utilizan la recursión, son los algoritmos divide y vencerás y los algoritmos de vuelta atrás.

### LA NATURALEZA DE LA RECURSIVIDAD:

Una función recursiva es aquella que se llama a sí misma, bien directamente o bien indirectamente a través de otra función. La recursividad es un tópico importante examinado frecuentemente en cursos en los que se trata de resolución de algoritmos y en cursos relativos a Estructura de Datos.

Una función que tiene sentencias entre las que se encuentra al menos una que llama a la propia función se dice que es recursiva. Así, la organización recursiva de una función `funcion1()` sería la siguiente:

```
Void función1(...) {  
...  
funcion1();    //llamada recursiva  
...  
}
```

### Ejemplo 1:

Planteamiento recursivo de la función matemática que suma los N primeros números enteros positivos.

Como punto de partida se puede afirmar que para  $n = 1$  se tiene que la suma  $S(1) = 1$ . Para  $n = 2$  se puede escribir  $S(2) = S(1) + 2$ ; en general y aplicando la inducción matemática se tiene:

$$S(n) = S(n-1) + n$$

El algoritmo que determina la suma de modo recursivo ha de tener presente un caso base y un caso recursivo. Así, en el caso del cálculo de  $S(6)$ , la definición es  $S(6) = 6 + S(5)$ , que de acuerdo a la definición es  $5 + S(4)$ , este proceso continúa hasta  $S(1) = 1$  por definición.

### Ejemplo 1:

En consecuencia, la implementación del algoritmo mencionado, que calcula la suma de los n primeros enteros:

```

long sumaNEnteros(int n) {
    if(n==1){
        return 1;
    }
    Else{
        return n + sumaNEnteros(n-1);
    }
}

```

Ejemplo 2:

Entonces se puede establecer que:

$\text{fibonacci}(0) = 0$

$\text{fibonacci}(1) = 1$

...

$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$

Y la definición recursiva será:

$\text{fibonacci}(n) = n$

$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$

Sí  $n=0$  o  $n=1$

Sí  $n \geq 2$

Así, por ejemplo:

$\text{fibonacci}(6) = \text{fibonacci}(5) + \text{fibonacci}(4)$

EJEMPLO 2:

En consecuencia, la implementación del algoritmo mencionado de la serie fibonacci es:

```

long fibonacci( int n){
    if(n == 0 || n == 1){
        return n;
    }
    Else{
        return fibonacci(n-1) + fibonacci(n-2)
    }
}

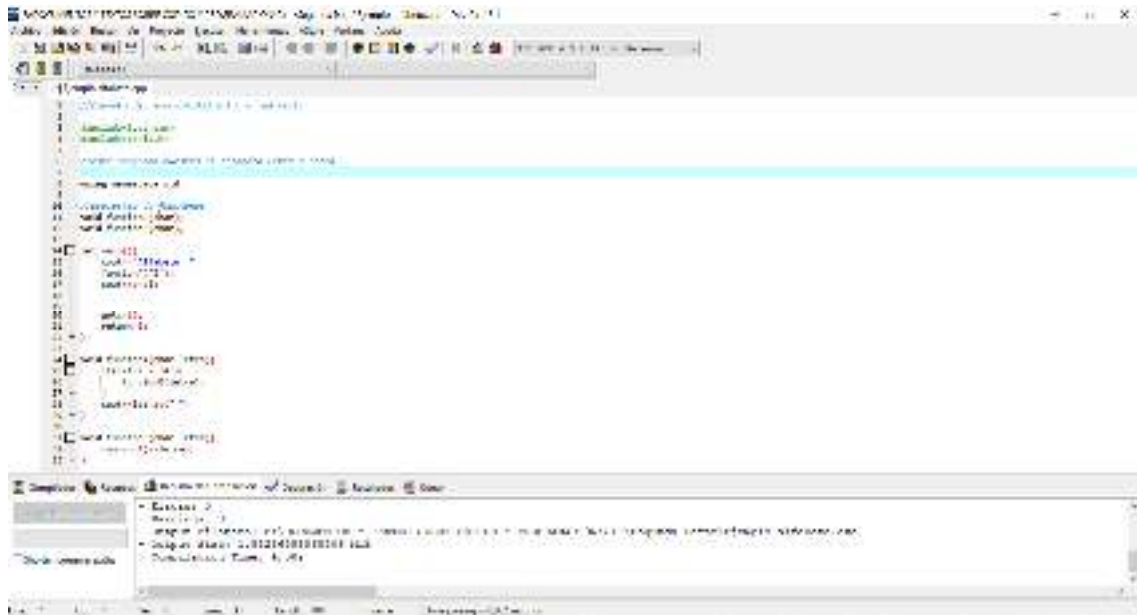
```

## RECURSIVIDAD INDIRECTA -----FUNCIONES MUTUAMENTE RECURSIVAS

La recursividad indirecta se produce cuando una función llama a otra, que eventualmente terminará llamando de nuevo a la primera función.

### EJEMPLO:

Hacer un programa que muestre el alfabeto, carácter a carácter, utilizando recursividad mutua o indirecta.



## RECURSIÓN VS ITERACIÓN

En las secciones anteriores se han estudiado varios algoritmos que se pueden implementar fácilmente de modo recursivo, o bien de modo iterativo. En este vídeo se comparan los dos enfoques y se examinan las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

(A partir de 2:01:28) [https://www.youtube.com/watch?v=MZiY4G\\_b2MQ](https://www.youtube.com/watch?v=MZiY4G_b2MQ)

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de llamada a función y, en consecuencia, se necesita un tiempo suplementario para realizar cada llamada. Esta característica puede resultar cara en tiempo de procesador y espacio de memoria.

Entonces **¿cuáles son las razones para elegir la recursión?** La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo.

### EJEMPLO:

Dado un número natural N, obtener la suma de los dígitos de que consta. Presentar un algoritmo recursivo y otro iterativo.



Directrices en la toma de decisión iteración/recursión:

- \*Considérese una solución recursiva sólo cuando una solución iterativa sencilla no sea posible.
- \*Utilícese una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.
- \*Sí son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a las funciones.
- \*En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fiables de leer y comprender que su correspondiente iterativa. En estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el coste extra (en tiempo y memoria) de la ejecución de programa recursivo.

#### ALGORITMO DIVIDE Y VENCERÁS

Una de las técnicas más importantes para la resolución de muchos problemas de computadora es la denominada “*divide y vencerás*”. El diseño de algoritmos basados en esta técnica consiste en transformar (dividir) un problema de tamaño **n** en problemas más pequeños, de tamaño menor que **n**, pero similares al problema original. De modo que resolviendo los subproblemas y combinando las soluciones se pueda construir fácilmente una solución del problema completo (vencerás).

Normalmente, el proceso de división de problemas en otros de tamaño menor va a dar lugar a que se llegue al **caso base**, cuya solución es inmediata. A partir de la obtención de la solución del problema para el caso, se combinan soluciones que amplían el tamaño del problema resuelto, hasta que el problema original queda resuelto.

Por ejemplo: plantea el problema de dibujar un segmento que está conectado por los puntos en el plano (x1, y1) y (x2, y2).

Un algoritmo “*divide y vencerás*” se define de manera recursiva, de tal modo que se llama a sí mismo sobre un conjunto menor de elementos. Normalmente, se implementan con dos



llamadas recursivas, cada una con un tamaño menor, generalmente la mitad. Se alcanza el caso base cuando el problema se resuelve directamente.

Consta de dos partes:

- 1) Dividir recursivamente el problema original en subprogramas cada vez más pequeños.
- 2) Resolver el problema dando solución a los subproblemas a partir del caso base.

### **TORRES DE HANOI**

El problema en cuestión supone la existencia de tres varillas (A, B, C) en los que se alojan discos que se pueden trasladar de una varilla a otra con una condición: cada disco es ligeramente inferior en diámetro al anterior.

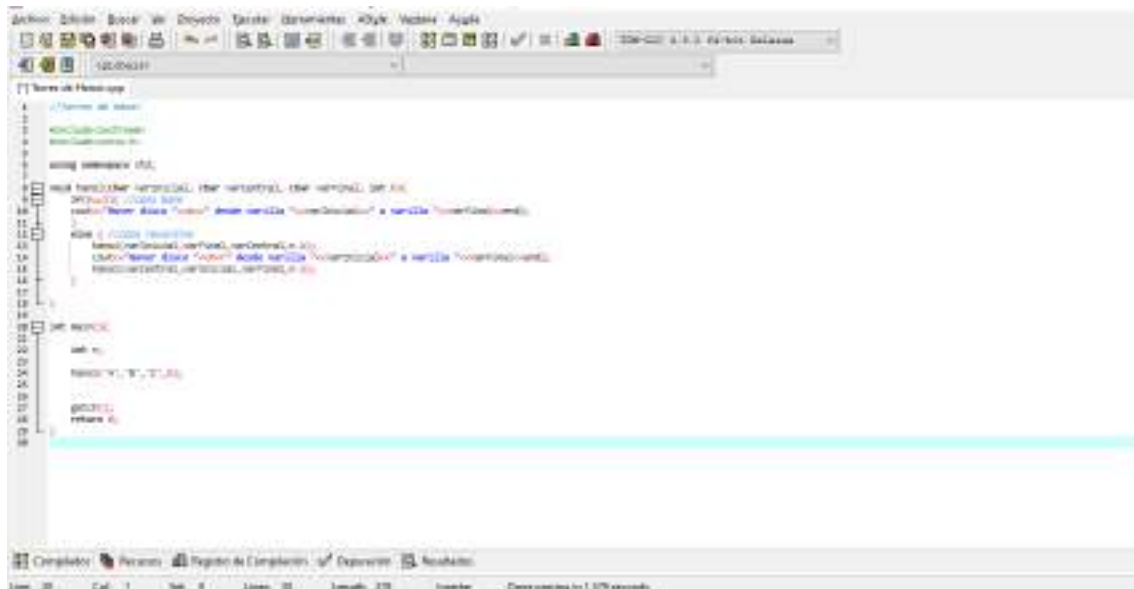
#### **Algoritmo recursivo para resolver las Torres de Hanoi**

```
Void hanoi(char varInicial, char varCentral, char varFinal, int n) {  
    If(n==1) { //caso base  
        Cout<<"Mover disco"<<n<<" desde varilla"<<varInicial<<" a varilla  
        "<<varFinal<<endl;  
    }  
    Else { //caso recursivo  
        hanoi(varInicial,varFinal,varCentral, n-1);  
        cout<<"Mover disco "<<n<<" desde varilla"<<varInicial<<" a varilla"<<varFinal<<endl;  
        hanoi(varCentral,varInicial,varFinal, n-1);  
    }  
}
```

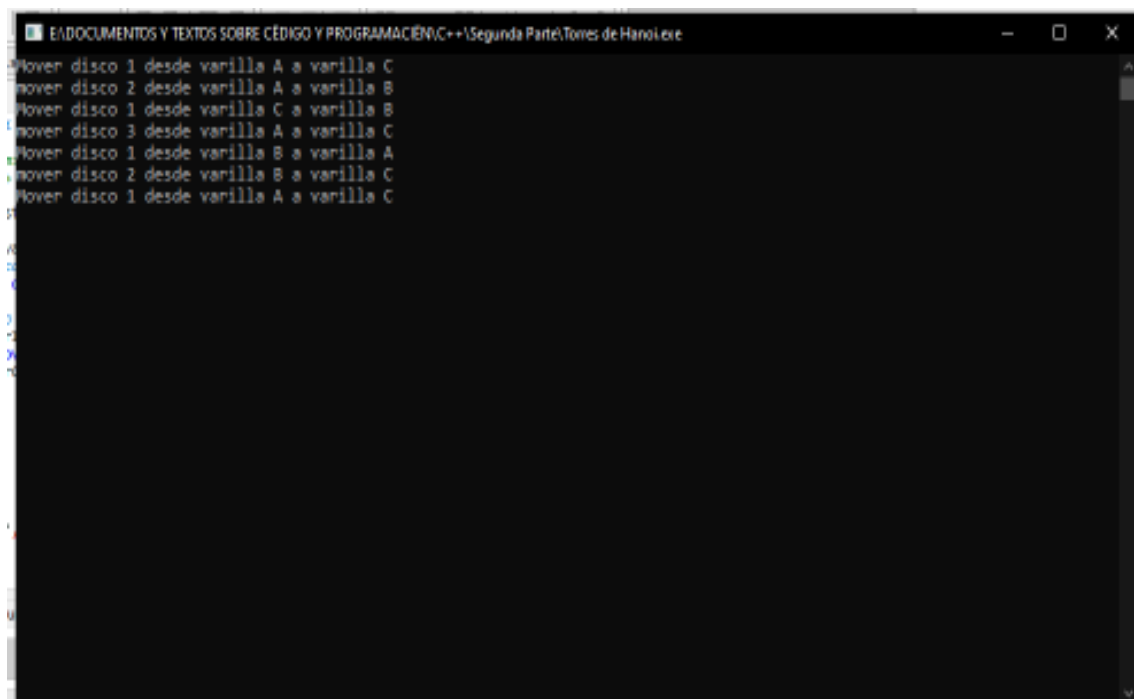
#### **Ejemplo en C++ del código y su ejecución**

<https://www.ajedrezeureka.com/torres-de-hanoi/>

link para comprobar el funcionamiento del código



```
1 //Towers of Hanoi
2
3 #include <iostream>
4 using namespace std;
5
6 void torresDeHanoi(int n, char varilla1, char varilla2, char varilla3, int k)
7 {
8     cout << "Mover disco " << k << " desde varilla " << varilla1 << " a varilla " << varilla3 << endl;
9 }
10
11 void torresDeHanoi(int n, char varilla1, char varilla2, char varilla3, int k)
12 {
13     if (n == 1)
14     {
15         torresDeHanoi(n, varilla1, varilla2, varilla3, k);
16     }
17     else
18     {
19         torresDeHanoi(n-1, varilla1, varilla3, varilla2, k);
20         torresDeHanoi(n, varilla1, varilla2, varilla3, k);
21         torresDeHanoi(n-1, varilla2, varilla1, varilla3, k);
22     }
23 }
24
25 int main()
26 {
27     int n;
28     char v1, v2, v3;
29     cout << "n: ";
30     cin >> n;
31     cout << "v1: ";
32     cin >> v1;
33     cout << "v2: ";
34     cin >> v2;
35     cout << "v3: ";
36     cin >> v3;
37     torresDeHanoi(n, v1, v2, v3, 1);
38     return 0;
39 }
```



```
1 Mover disco 1 desde varilla A a varilla C
2 Mover disco 2 desde varilla A a varilla B
3 Mover disco 1 desde varilla C a varilla B
4 Mover disco 3 desde varilla A a varilla C
5 Mover disco 1 desde varilla B a varilla A
6 Mover disco 2 desde varilla B a varilla C
7 Mover disco 1 desde varilla A a varilla C
```

### ¿Cómo se calcula un numero par?

Lo mejor para estos casos es considerar el 0 como elemento principal; primero comparamos si el número ingresado es igual a cero, a partir de aquí es donde calculamos la paridad con la función MOD. La paridad de un numero (o su imparidad) en pseint se puede obtener con una función llamada MOD o resto y un condicional. Se hace de la siguiente forma: en el condicional (if o si) se agrega como condición esto -> (num % 2) ==0. Donde num es la variable numérica entera, MOD % es lo que se conoce como dividir un número (num) por dos y que su resto sea 0, entonces se cumple con la idea de paridad. Se utilizan dos signos iguales ( = = ) para comparar el resultado de la división. Sí usamos uno solo, estamos asignando el valor 0 en lugar de compararlo. En síntesis, estamos comparando tres sentencias: si el número es cero, si es par o

sí es impar. Si no contamos el cero como elemento comparativo el mismo se incluirá en los números pares.

### **¿Cómo usa la variable “contador”?**

“Contador” es una variable cuyo valor se incrementa o decrementa en una cantidad constante cada vez que se produce un determinado suceso o acción. Una vez creada la variable “contador”, se inicia con algún valor antes de usarla en algún bucle. Por lo general se inicializa en 0 pero también puede ser 1 o depende de la situación. Una vez dentro del bucle, antes de salir de la iteración hay que actualizar el valor del “contador”, se hace de la siguiente manera: contador = contador + 1. Se suele usar para contar elementos de manera individual, acumularlos y mostrar la cantidad final.

### **¿Cómo se usa la función “Mientras”?**

La función “Mientras” presenta la particularidad de que para entrar en el bucle la condición tiene que ser verdadera desde el principio (en la mayoría de los casos). Antes de usarla hay que inicializar la variable que usamos en el bucle como verdadera o con los parámetros necesarios. Al ser verdadera o true se ingresa al loop, pero de ser falsa o false se esquivo la función por lo que sigue con el resto del código.

### **¿Cómo se crea un arreglo?**

En el principio del código definimos el arreglo con el tipo de dato que va a contener. Luego definimos la dimensión, este es el tamaño o cantidad de elementos a conservar. Con un bucle “para” podemos llenar el arreglo, recordando que los sub índices empiezan (generalmente) en 0 o desde la posición asignada. Leemos el arreglo con el sub índice [i] (el más común) dentro del “para” antes de finalizarlo.

Para mostrar el arreglo utilizamos otro bucle “Para” que en lugar de llenar los sub índices, los muestre con la función mostrar o escribir. En otros lenguajes los arreglos tienen distintos sub índices [i] o [k] o [j] dando a entender que son distintos arrays, por lo que para llenarlo o mostrarlo hay que especificar el arreglo a utilizar con su correspondiente sub índice.

### **¿Cómo se crea una matriz?**

Comenzamos definiendo columnas, filas y matriz como enteros (o con el tipo de dato necesario). Luego definimos la variable a introducir en el “para” y la dimensión. Si la dimensión es variante, primero se pide la dimensión y luego se la define. Ahora abrimos un bucle “para” con filas desde valor inicial hasta dimensión menos 1 y con paso uno; en la siguiente línea abrimos otro bucle para llenar las columnas igual al anterior. Pedimos por teclado el ingreso de un valor y leemos esa variable. Asignamos la matriz con matriz[filas, columnas] <- num. Para mostrarla copiamos el mismo bucle doble y escribimos por pantalla la matriz. Su traspuesta es la misma forma pero cambiando el orden [filas, columnas] por [columnas, filas]. Para sumar dos matrices necesitamos una tercer matriz en la cual almacenamos las anteriores de la siguiente manera: MatrizResultado[filas, columnas] = matriz[filas,columnas] + matriz[columnas,filas] (en este caso guardo en MatrizR la suma de una matriz y su traspuesta).

Algoritmo Ejercicio\_Propio\_Matrices

definir columnas, filas, matriz como entero;

definir num, dim como entero;

```

escribir "Ingrese una serie de valores enteros para llenar la matriz";
escribir "Antes vamos a definir la dimension";
escribir "Ingrese la dimension";
leer dim;
dimension matriz[dim,dim];
filas <- 0;
columnas <- 0;
num <- 0;

para filas <- 0 hasta dim -1 con paso 1 Hacer
    para columnas <- 0 hasta dim -1 con paso 1 Hacer
        escribir "Ingrese un numero";
        leer num;
        matriz[filas, columnas] <- num;
    FinPara
FinPara
escribir "Matriz Original";
para filas <- 0 hasta dim -1 con paso 1 Hacer
    para columnas <- 0 hasta dim -1 con paso 1 Hacer
        escribir matriz[filas, columnas], " | " sin saltar;
    FinPara
    escribir"";
FinPara
escribir "Matriz Trapuesta";
para filas <- 0 hasta dim -1 con paso 1 Hacer
    para columnas <- 0 hasta dim -1 con paso 1 Hacer
        escribir matriz[columnas, filas], " | " sin saltar;
    FinPara
    escribir"";
FinPara
escribir "Ahora vamos a sumar la original con la traspuesta";

```

```

definir matrizR como entero;
dimension matrizR[dim, dim];

para filas <- 0 hasta dim -1 con paso 1 Hacer
    para columnas <- 0 hasta dim -1 con paso 1 Hacer
        matrizR[filas,columnas] <- matriz[filas,columnas] +
matriz[columnas,filas];
    FinPara
FinPara
escribir "Resultado";
para filas <- 0 hasta dim -1 con paso 1 Hacer
    para columnas <- 0 hasta dim -1 con paso 1 Hacer
        escribir matrizR[filas,columnas], " | " sin saltar;
    FinPara
    escribir "";
FinPara
escribir "Y su traspuesta";
para filas <- 0 hasta dim -1 con paso 1 Hacer
    para columnas <- 0 hasta dim -1 con paso 1 Hacer
        escribir matrizR[columnas,filas], " | " sin saltar;
    FinPara
    escribir "";
FinPara
FinAlgoritmo

```

### **¿Cómo se hace una tabla de Multiplicar variante?**

Se definen tres variables: array, num y limit. Array para la colección de datos. Num es el número del cual vamos a mostrar la tabla. Limit es el límite o cantidad de elementos a multiplicar. Imprimimos por pantalla y pedimos los datos. Creamos la dimensión del arreglo después de tener los datos enteros (sí lo hacemos antes puede dar error). Como la dimensión tiene que ser variable le asignamos el valor de limit entre [ ]. Queda así: Dimensión array[limit]. Con un bucle “para” calculamos la tabla para toda la colección, donde el número por la cantidad de iteraciones va a ser el resultado del arreglo. La tabla final se puede mostrar con otro bucle “para” con

detalles o sin ellos. Para no quedar fuera de rango, al limit o destino de la función “para” se le resta 1.

### ¿Cómo se crea un arreglo y su invertido?

Definimos dos arreglos como enteros (en este caso). Definimos dos variables: una para la dimensión y otra para el bucle. Agregamos un índice “j” para usar más adelante. Pedimos por pantalla la longitud del arreglo y de ahí obtenemos la dimensión, que la declaramos después de este dato para no tener un error. Ahora procedemos a llenar el bucle de manera normal con un bucle “para” ->

para i<-0 hasta num-1 con paso 1 Hacer

```
    escribir "Ingrese un numero entero";
```

```
    leer num1;
```

```
    array1[i] <- num1;
```

FinPara

Con otro bucle lo mostramos: ->

```
    escribir sin saltar array1[i], " | ";
```

FinPara

```
    escribir"";
```

Ahora, usamos el sub índice “j” para invertir el arreglo original. Se utiliza un “para” pero de forma inversa. Se inicia desde el valor más alto y su destino es el valor 0, con paso de -1.

Es decir que el sub índice “j” comienza desde el valor más alto (el usado para llenar el primer arreglo con un factor de corrección para no quedar fuera de lugar con los índices, por lo tanto, le restamos 1) hasta el valor más chico que es el 0. El paso -1 nos da la idea de que vamos a retroceder en lugar de avanzar. ¿Por qué J? porque necesitamos el array [i] pero sin llenar (o vacío de estructura si así lo prefieren), de esta manera tomamos el original y lo trasladamos al segundo, pero con un orden inverso.

Al segundo arreglo le asignamos el primero, pero con diferente sub índice. Para mostrarlo por pantalla es el mismo procedimiento. Debería quedar de la siguiente forma:

->

para j <- (num-1) hasta 0 con paso -1 Hacer

```
    array2[j] <- array1[j];
```

FinPara

```
    escribir "Arreglo 1 invertido";
```

para j <- (num-1) hasta 0 con paso -1 Hacer

```
escribir sin saltar " | " , array2[jj];
```

```
FinPara
```

```
escribir"";
```

### ¿Cómo se usa e invoca una función?

Una función sirve para complementar el código principal. Nuestro algoritmo principal o main queda más prolijo cuando usamos funciones o sub procesos. Para crear la función lo que debemos hacer es: fuera del main (principal) colocar la palabra función o sub algoritmo. Tenemos tres partes a considerar: la variable de retorno, el nombre de la función y los argumentos. En algunos casos alguna de estas puede no estar o estar en blanco o vacía. Su estructura queda de la siguiente manera: ***función variable de retorno <- nombre(argumentos)***. El ejemplo típico es este:

```
función sumar <- dosvalores(num1 por referencia, num2 por referencia)
```

```
    definir sumar como real;
```

```
    sumar <- (num1 + num2);
```

```
FinFuncion
```

Lo que deducimos es que el nombre de la función es “dosvalores” y su variable de retorno es “sumar”. Los argumentos son “num1” y “num2” por referencia. Para no entrar en confusión, en la función no definimos nada que no sea necesario, es decir que no creamos una variable llamada “resultado” y le asignamos la suma de num1 y num2, sino que la suma de estas variables ya mencionadas debe asignarse al retorno “sumar”

Todo esto en el código principal se invoca con una variable, a la cual le asignamos el nombre de la función (dosvalores) y pasándole como parámetros entre paréntesis las variables del código principal. Queda de la siguiente manera: `retorno <- dosvalores(n1, n2);`

```
funcion sumar <- dosvalores(num1 por referencia, num2 por referencia)
```

```
    definir sumar como real;
```

```
    sumar <- (num1+num2);
```

```
FinFuncion
```

Algoritmo Sub\_proceso\_Vuno

```
    definir n1 como real;
```

```
    definir n2 como real;
```

```
    definir retorno como real;
```

```
    escribir "Ingrese un numero";
```

```
    leer n1;
```

```
escribir "Ingrese un segundo número";
```

```
leer n2;
```

```
retorno <- dosvalores(n1,n2);
```

```
escribir "El resultado de la suma de ",n1," y ",n2," es: ",retorno;
```

```
escribir "El resultado de la suma de ",n1," y ",n2," es: ",dosvalores(n1,n2);
```

Esta última forma también es válida.

FinAlgoritmo

### **Análisis de un ejemplo de recursión**

En el ejemplo, queremos sumar dos valores y lo que hay entre ellos, por ejemplo 1 y 5. La primera forma puede ser con “para” dentro del código principal, estableciendo cual es el límite inferior y cual el superior. Iniciamos el “Para” en el inferior hasta el superior, en un acumulador suma (por ejemplo) vamos sumando lo que tiene la variable más el iterador ( i ); en la siguiente línea escribimos el iterador o después del bucle. Todo esto también se puede plasmar en una función. Ahora viene la Recursión, tenemos dos variables (x1 y x2) y una tercera ( sum) para acumular los resultados. Una variable se usará de referencia, funcionará como iterador (x1) y la otra (x2) como elemento de comparativa. Sum va a almacenar todos los resultados del iterador (x1). Creamos un SI en el cual si  $x2 = x1$  el resultado es  $sum+x2$  lo cual constituye el caso base o el corte en la recursión para no quedar en un loop infinito. Hasta no cumplirse el SINO hará uso del acumulador sum y del iterador (x1). Suma será  $sum + x1$  y  $x1$  será  $x1 + 1$  y en la siguiente línea llamamos a la función estableciendo la recursión.

```
funcion sumar <- valores(n1 por referencia, n2 por referencia)
```

```
escribir "Los valores son: ",n1," y ",n2;
```

```
para i <- n1 hasta n2 con paso 1 Hacer
```

```
escribir " ** ",i," ** ";
```

```
escribir "Sumando ";
```

```
resultado <- resultado + i;
```

```
escribir resultado;
```

```
FinPara
```

```
escribir "El resultado por FUNCION es: ",resultado;
```

FinFuncion



### Algoritmo Sumas\_sucesivas

definir num1, num2 como entero;

definir limInf, limSup como entero;

escribir "Ingrese dos numeros enteros diferentes entre si";

escribir "Ingrese el primer numero";

leer num1;

escribir "Ingrese el segundo numero";

leer num2;

si (num1 > num2) Entonces

    limInf <- num2;

    limSup <- num1;

    escribir "Destino: ", limSup;

    escribir "Origen: ", limInf;

SiNo

    si (num2 > num1) Entonces

        limInf <- num1;

        limSup <- num2;

        escribir "Destino: ", limSup;

        escribir "Origen: ", limInf;

    FinSi

FinSi

definir suma como entero;

suma <- 0;

dif <- (limSup - limInf)+1;

para i <- limInf hasta limSup con paso 1 Hacer

    escribir " | ", i, " | ";

    escribir "Sumando ";

```
suma <- suma + i;
```

```
escribir suma;
```

FinPara

```
escribir "La suma entre ",limInf," y ",limSup," es: ",suma;
```

```
escribir "El mismo proceso pero por FUNCION";
```

```
retorno1 <- valores(limInf,limSup);
```

```
escribir retorno1;
```

```
escribir "";
```

```
escribir "El mismo proceso pero por RECURSION";
```

```
retorno2 <- recursion(limInf,limSup,0);
```

```
escribir retorno2;
```

```
escribir "";
```

```
escribir "El mismo proceso pero por RECURSION atras";
```

```
retorno3 <- recursion2(limInf,limSup,0);
```

FinAlgoritmo

```
funcion recursar <- recursion(x1,x2, sum )
```

```
si x2 == x1 entonces
```

```
escribir "El resultado por recursion es: ",sum + x2; //caso base
```

SiNo

```
sum <- sum + x1; //acumulador
```

```
x1 <- x1+1; //iterador
```

```
recursar <- recursion(x1,x2, sum ); //llamamos a la funcion
```

FinSi

FinFuncion

```
funcion recursar2 <- recursion2( x1, x2, sum2)
```

si x2 == x1 entonces

escribir "El resultado por recursion 2 es: ",sum2 + (x2); //caso base

SiNo

sum2 <- sum2 + x2; //acumulador

x2 <- x2 -1; //iterador

recursar2 <- recursion2(x1,x2,sum2); //llamamos a la funcion

FinSi

FinFuncion

### **Estructura repetitiva FOR**

Estructura repetitiva con la cual podemos ejecutar unas instrucciones un número finito de veces, a diferencia del while que se repite una cantidad de veces según la condición se cumpla, el FOR solo se la da el número de veces que se quiere repetir.

### **Estructura repetitiva SWITCH CASE**

En los lenguajes de programación, un switch case es una declaración de interrupción. Es un tipo de mecanismo de control de selección utilizado para permitir que el valor de una variable o expresión cambie el flujo de control de la ejecución del programa mediante búsqueda

### **Función y Variable de Retorno**

Es una variable que nos devuelve el valor de una función, esto quiere decir que el resultado de todas las operaciones hechas en una función quedara almacenado en dicha variable. Si queremos utilizar la función en nuestro algoritmo debemos saber que siempre que invoquemos la función nos va a devolver el mismo tipo de dato de la variable, o lo que es igual: la función recibe un dato de tipo entero y devuelve (o retorna) un entero (nunca un booleano o un real).

### **Vectores**

Es una zona de almacenamiento contiguo, que contiene una serie de elementos del mismo tipo y se distinguen entre sí por un índice y un nombre.

### **Matrices**

Una matriz es un espacio bídimensional que se asigna en la memoria del computador. Las matrices tienen un nombre que por lo general está dado por una letra del alfabeto o combinación de ellas. También se debe definir el tipo de dato que va a almacenar. Una matriz no puede almacenar un dato de cada tipo, si puede haber distintas matrices de cada tipo de dato. De igual forma que los vectores a estas hay que definir las en tamaño pero cada valor indica fila y columna, ejemplo: [3,3] o [3,4].