



APRENDIENDO GARGAR



VERSION 1.0 - 2012

Tabla de contenido

1.	<i>GarGar, el lenguaje de programación.....</i>	7
1.1	¿Qué es GarGar?.....	7
1.2	Palabras Reservadas	8
2.	<i>Hola mundo en GarGar</i>	9
2.1	Introducción a los procedimientos (¿Tanto para mostrar un “hola mundo”?)	9
2.2	Introducción a las sentencias en un programa.....	10
2.3	¿Y qué es eso de que me obligás a declarar PRINCIPAL y SALIDA? .	11
3.	<i>Variables</i>	13
3.1	¿Qué es una variable?	13
3.2	Tipos de variable que existen (Porque no todas las variables son del mismo tipo).....	13
3.3	¿Y cómo le coloco un valor a una variable?.....	14
3.4	¿Qué pasa si mezclo los tipos de dato?	16
3.5	¿Y para que me sirve asignarle un valor a una variable?	16
3.6	Operaciones entre variables (¡Ahora entiendo para que sirven!)...	17
3.7	¿Qué es un arreglo? (o como un arreglo se parece sospechosamente a un montón de variables juntas).....	22
3.8	Operaciones entre arreglos	23
4.	<i>Constantes.....</i>	25

4.1	¿Qué es una constante? (o la historia de la variable que no se dejaba modificar).....	25
4.2	¿Y si no se pueden modificar, en que se parecen a las variables? (o como las constantes se copian de los tipos de las variables)	25
4.3	¿Y para que me sirven las constantes?	26
4.4	¿Y qué pasa si yo quiero asignarle un nuevo valor a una constante? 27	
5.	<i>Estructura de un programa GarGar.....</i>	29
5.1	Espacio de declaraciones globales (hogar de las variables globales y las constantes)	29
5.2	Variables globales vs Variables locales (¿Quién es más importante?) 31	
5.3	¿Existen también los arreglos como variable global?.....	32
5.4	¿Y las constantes globales?.....	32
5.5	Espacio de declaraciones de procedimientos (¡el orden es importante!)	33
6.	<i>Procedimientos</i>	37
6.1	¿Qué es un procedimiento? (¡Decime por favor, para que lo puedo usar!) 37	
6.2	¿Me gusta el nombre “proc” para mis procedimientos, lo puedo usar para todos?	38
6.3	¿Me recordás lo de las variables locales?.....	38
6.4	Parámetros de un procedimiento (¿o más tipos de variables?).....	39
6.5	Tipos de parámetros (por dios, cuantos tipos hay)	40

6.6	¿Cómo uso un procedimiento?	42
6.7	El restrictivo procedimiento SALIDA.....	42
7.	<i>Funciones.....</i>	43
7.1	¿Qué es una función? (se ve como un procedimiento con un tipo de dato) 43	
7.2	¡Quiero declarar una función! (explicame de una vez como hago) 45	
7.3	¡Que bueno, no veo la hora de declarar mi función que retorne un arreglo!	46
7.4	¿Y para que me sirve una función?.....	46
8.	<i>Sentencias del lenguaje GarGar.....</i>	47
8.1	ASIGNACION: La sentencia para colocarle valor a las variables.	48
8.2	LLAMAR: La sentencia para llamar a un procedimiento.....	49
8.3	MOSTRAR: La sentencia de impresión en pantalla.....	50
8.4	LEER: La sentencia de lectura de datos por teclado	51
8.5	MIENTRAS: La estructura iteradora (iteradora, iteradora, iteradora...)	52
8.6	SI: La estructura condicional (¿es esta o la otra?)	54
9.	<i>Ejemplos de programas completos</i>	57
9.1	Ejemplo 1: Algoritmo de Ordenamiento por Burbujeo	57
9.2	Ejemplo 2: Algoritmo de Raíz cuadrada por aproximación	59
10.	<i>Funciones ya provistas por el lenguaje GarGar</i>	61
10.1	Función EsImpar	61

10.2	Función EsPar.....	62
10.3	Función Potencia.....	63
10.4	Función Raiz	64
10.5	Función Redondear.....	65
10.6	Función Truncar	66
10.7	Función ValAbs.....	67
10.8	Función Pi.....	68
10.9	Función Seno.....	69
10.10	Función rSeno	69
10.11	Función Coseno.....	70
10.12	Función rCoseno	70
10.13	Función Tangente	71
10.14	Función rTangente	71
11.	<i>Buenas prácticas del lenguaje GarGar.....</i>	73

1. GarGar, el lenguaje de programación

1.1 *¿Qué es GarGar?*

GarGar es un lenguaje simple y en castellano, con un formato pseudocódigo. Está enfocado a brindar los primeros pasos en el mundo de la programación.

GarGar sigue un enfoque estructurado, de lectura de código de arriba a abajo, izquierda a derecha, emulando la manera natural de lectura que tiene una persona.

GarGar elige en todos los casos la simpleza, sin dejar de lado la robustez y la flexibilidad del lenguaje. Por esta razón, no se encuentran más que una sola estructura iteradora y condicional en el lenguaje.

1.2 Palabras Reservadas

GarGar como todo lenguaje de programación tiene palabras reservadas. Estas palabras y letras ya tienen significados dentro del lenguaje GarGar. A lo largo del tutorial las iremos viendo en detalle.

Op. de Numero	Iteradores	Tipos de Dato	Decl. globales
+	mientras	texto	variables
-	hacer	numero	constantes
*	finmientras	booleano	
/			Tipos
	Condicionales	Tipos de Variable	arreglo
Op. Booleanos	si	var	
<	entonces	const	Const. Booleanas
<=	sino		verdadero
>	finsi	Interacción Usuario	falso
>=		leer	
=	Decl. rutinas	mostrar	Op. Booleanos
<>	procedimiento		and
!	finproc	Otros	or
	funcion	;	
Op. Texto	finfunc	:	Procs especiales
&	comenzar	,	Salida
	llamar	(principal
Comentarios)	
{	Asignacion	de	
}	=	ref	

2. Hola mundo en GarGar

2.1 *Introducción a los procedimientos (¿Tanto para mostrar un “hola mundo”?)*

Como primer paso en el camino a dominar el lenguaje GarGar, comenzaremos con el paso más básico y repetido de cualquier tutorial en cualquier lenguaje de programación: mostrar en pantalla la leyenda “Hola mundo”. Si quisiéramos que en nuestra pantalla aparezca esa frase, basta con colocar el siguiente fragmento de código:

```
procedimiento SALIDA ( )  
  comenzar  
    Mostrar('Hola mundo');  
  finproc ;  
  
procedimiento PRINCIPAL ( )  
  comenzar  
    llamar SALIDA ( ) ;  
  finproc ;
```

¿Parece excesivo para mostrar una sola línea en pantalla no? ¿Mentimos acaso cuando dijimos que era un lenguaje simple? Nada de eso. Si bien es cierto que son varias líneas de código para una sola función, tiene una explicación, y es una de las grandes fortalezas de GarGar.

Si uno es medianamente observador, podrá haberse dado cuenta que hay 2 palabras en mayúscula. Esas palabras, en ambos casos están precedidas por la palabra procedimiento. Y no solo eso, **comenzar**, y **finproc** también se repiten. Sin embargo, lo que está entre **comenzar** y **finproc** es distinto.

Lo que se está haciendo en este programa es declarar 2 procedimientos, uno de ellos llamado SALIDA, y otro de ellos llamado PRINCIPAL. Dentro de SALIDA, se colocó que muestre “Hola Mundo”, mientras que PRINCIPAL tiene la sentencia que llama al procedimiento SALIDA. Las palabras **comenzar** y **finproc** son los que delimitan donde empieza y dónde termina el cuerpo de cada procedimiento (Cuerpo de un procedimiento es aquel espacio en donde se colocan las sentencias).

¿Vieron que era simple?

Ya vamos a ir más en detalle en procedimientos en capítulos más adelante.

Pero lo que sí es importante que sepas ahora, es que estos dos procedimientos aparecen en cualquier programa y son obligatorios. Ningún programa podrá existir sino tiene definidos estos dos procedimientos en este orden.

2.2 Introducción a las sentencias en un programa

Recién en el ejemplo hablábamos de que declaramos 2 procedimientos. La palabra “declarar” es una palabra que se usa frecuentemente en programación. Dentro de estas declaraciones de procedimientos, lo que hicimos fue usar sentencias propias del lenguaje GarGar para lograr que nuestro programa muestre en pantalla “Hola Mundo”.

En este caso una de las sentencias que usamos es “mostrar”, que lo que hace es imprimir en pantalla lo que coloquemos entre paréntesis. No te preocupes si no lo entendés ahora, cada sentencia tiene sus particularidades que vamos a ir viendo a lo largo del manual. Lo importante ahora es entender que un programa está hecho de

declaraciones de procedimientos, y que estos procedimientos dentro de ellos tienen sentencias.

La otra sentencia que usamos es la sentencia llamar, que lo que hace es llamar a un procedimiento ya declarado. En este caso, dentro de principal, lo que hicimos fue llamar al procedimiento salida. Llamar a otro procedimiento es decirle en una sola sentencia “hace todo lo que hace este procedimiento ahora”. Vamos a entrar más en detalle en esto más adelante también.

2.3 ¿Y qué es eso de que me obligás a declarar PRINCIPAL y SALIDA?

Ambos procedimientos deben estar declarados en todos los programas que se escriban en lenguaje GarGar por distintas razones.

PRINCIPAL es el procedimiento de entrada del programa. La primer línea debajo del comenzar de este procedimiento es la primer línea que se ejecutará de tu programa. El procedimiento PRINCIPAL es la manera que uno tiene para explicarle al compilador que la ejecución del programa debe empezar por ahí.

SALIDA, por otra parte, existe por una razón más compleja. SALIDA es el procedimiento es donde uno debería mostrar los datos finales del programa. Por ejemplo, si nuestro programa lo que hace es calcular el promedio de un alumno, la impresión del resultado en pantalla debería estar en el cuerpo de este procedimiento. Nadie te obliga a que lo hagas, pero es una buena práctica de cualquier lenguaje de programación, separar los resultados finales, del cálculo de los mismos.

Además, el correcto uso del procedimiento SALIDA, posibilita el uso en el aplicativo Ragnarok, de los tests de prueba.

3. Variables

3.1 *¿Qué es una variable?*

La mejor manera de explicar que es una variable, es imaginarla como un recipiente: En una variable se guardan cosas. En realidad, en una variable se guardan valores. Se llaman variables, ya que su contenido puede cambiar en el transcurso del programa. Una variable puede estar conformada siempre por letras y números, y el carácter especial `_`. Forzosamente, siempre deben empezar por una letra.

Así se declara una variable en GarGar:

```
var mensaje : texto ;
```

Ahí está la declaración de una variable en GarGar. Ah, cierto. No hablamos del tipo de dato de la variable, que es esa palabra color rojo oscuro.

3.2 *Tipos de variable que existen (Porque no todas las variables son del mismo tipo)*

Si antes dijimos que una variable es un recipiente, el tipo de dato sería QUE cosas puede contener ese recipiente. Un colador es bueno para poner los fideos, pero no sirve para servir una gaseosa. Con las variables y sus tipos de datos es lo mismo. La variable que es de un tipo, no tolera que se le guarde un valor que no sea del mismo tipo con el cual se declaró.

Los tipos de dato son 3, y están en el siguiente ejemplo:

```
var mensaje : texto ;  
var calculo : numero ;  
var decision : booleano ;
```

¿Qué valores tolera cada uno entonces?

- El tipo de dato **texto** es auto descriptivo: Es para guardar texto.
- El tipo de dato **numero** es también auto descriptivo: Se usan para guardar números, tanto enteros como decimales.
- El tipo de dato **booleano**, representa únicamente 2 valores posibles: verdadero y falso. O sea que una variable declarada como booleana, solo podrá contener el valor verdadero, o el valor falso.

3.3 ¿Y cómo le coloco un valor a una variable?

La manera de colocarle un valor a una variable es mediante la sentencia asignación. Por esto es que comúnmente se dice que se le “asignan” valores a las variables. Miremos este ejemplo:

```
procedimiento ejemplo ( )  
var x : numero ;  
comenzar  
    x = 4 ;  
finproc ;
```

Este es otro procedimiento, llamado ejemplo (si, se pueden declarar procedimientos aparte de SALIDA y PRINCIPAL, pero lo vemos más adelante. Y si, no es necesario poner el nombre en mayúscula. Es más, no es necesario ni en SALIDA ni en PRINCIPAL, así que voy a dejar de ponerlos en mayúscula)

En este procedimiento se puede ver que declaramos una variable x, y más adelante, se ve que a la variable x se le asigna el valor 4. ¿Bastante simple no?

En este ejemplo vemos asignaciones para todos los tipos de dato:

```
procedimiento ejemplo ( )  
var x : numero ;  
var y : texto ;  
var z : booleano ;  
comenzar  
    x = 4.45 ;  
    y = 'Buenas tardes' ;  
    z = verdadero ;  
finproc ;
```

En este caso, estamos viendo como asignamos valores del tipo texto y booleano a las variables. En estos casos de ejemplo, estamos viendo como asignarle explícitamente un valor a una variable. En el caso de la variable y, le estamos asignando la frase buenas tardes, pero entre comillas simples. ¿Por qué es esto? Simple, porque esa es la manera que tenemos para decirle al compilador “Esto trátalo como un tipo de dato texto”. Si no se colocaran las comillas simples, sería imposible para el compilador GarGar saber que es un texto.

Para el caso de la variable z, le estamos asignando verdadero. En este caso, escribimos verdadero y GarGar ya sabe que se refiere al valor verdadero del tipo booleano.

3.4 ¿Qué pasa si mezclo los tipos de dato?

A GarGar no le gusta que se mezclen los tipos de dato. Si apropiado o sin querer asignas un valor que es de un tipo distinto al declarado en la variable, el compilador GarGar te arrojará un error.

```
procedimiento ejemplo ( )  
var x : numero ;  
comenzar  
    x = 'Buenas tardes' ; {Esto arroja error }  
finproc ;
```

En este caso x está declarado como numero, y se le está intentando asignar un texto. Esto no se puede, y el compilador te lo hará saber.

Para los curiosos: Lo que está a la derecha de la asignación entre llaves y en color verde, es lo que se llama un comentario. En un comentario se puede colocar lo que uno quiera, ya que el compilador GarGar lo ignorará por completo. Un comentario se abre y cierra con llaves.

3.5 ¿Y para que me sirve asignarle un valor a una variable?

Quizás no lo dije antes, pero una variable se puede asignar a otra, así que es lo que voy a hacer ahora (¡en este manual puedo hacer lo que quiero!) Cuando asigno una variable a otra, lo que estoy haciendo es copiando el contenido de una a otra.

```
procedimiento ejemplo ( )  
var x : numero ;  
var z : numero ;  
comenzar
```



```
x = 5 ;  
z = 15 ;  
x = y; {x ahora pasa a valer 15, y z sigue siendo 15 también }  
finproc ;
```

3.6 Operaciones entre variables (¡Ahora entiendo para que sirven!)

¡Me está faltando algo importante! GarGar permite realizar operaciones entre variables, entre valores explícitos, y operaciones entre ambos también.

Cada tipo de dato tiene su propio conjunto de operaciones posibles:

- Los números pueden ser sumados, restados, multiplicados y divididos
- Los textos únicamente pueden ser concatenados (o sea, agregar uno al final de otro)
- Los booleanos pueden ser operados con la suma booleana (como decir “esto o esto”) y la multiplicación booleana (como decir “esto y esto”)

Ejemplo de operaciones numéricas:

```
procedimiento ejemplo ( )  
var x : numero ;  
var z : numero ;  
comenzar  
    x = 5 ;  
    z = 15 ;  
    x = z + x ; {suma}  
    z = z * 2; {multiplicación}  
    x = z - x ; {resta}  
    z = z / 2; {división}  
finproc ;
```

Ejemplo de operaciones de texto:

```
procedimiento ejemplo ( )  
var x : texto ;  
var z : texto ;  
comenzar  
    x = 'Hola' ;  
    z = 'que tal' ;  
    x = z & x ; {concatenación}  
    z = z & 'chau'; { concatenación }  
finproc ;
```

Ejemplo de operaciones booleanas:

```
procedimiento ejemplo ( )  
var x : booleano ;  
var z : booleano ;  
var res : booleano ;  
comenzar  
    x = verdadero ;  
    z = falso ;  
    res = z or x ; {suma booleana} {res se le asigna verdadero}  
    res = z and x ; {multiplicación booleana} {res se le asigna falso}  
    res = z or verdadero ; {suma booleana} {res se le asigna verdadero}  
    res = z and verdadero ; {multiplicación booleana} {res se le asigna  
falso}  
finproc ;
```

Para dejar en claro: Lo que ocurre a la derecha de la asignación, se suele llamar expresión. Y también se suele llamar expresiones a los operandos de una operación. En otras palabras, una expresión es un conjunto de operaciones y operandos. Cada expresión, termina teniendo un tipo de dato, que es justamente del tipo del operador empleado. Ejemplo:

```
res = z or verdadero ;
```

El operador **or** es un operador entre booleanos, así que el compilador GarGar esperará que ambos operandos sean del tipo booleano. Y es más, el resultado de la operación es del tipo booleano (ya que **or** es un operador booleano), así que el compilador esperará que **res** sea una variable declarada como booleano.

Hay una mención especial que hacer para los booleanos, ya que existen ciertos operadores que son aplicables a cualquier tipo de dato, pero que

el resultado que devuelven, siempre es booleano. Estos son los casos de los operadores de comparación. Todos están listados en el siguiente ejemplo:

```
procedimiento ejemplo4 ( )
var x : booleano ;
var z : booleano ;
var res : booleano ;
comenzar
    res = 4 = 4 ; { igual } { res se le asigna verdadero }
    res = 4 <> 4 ; { distinto } { res se le asigna falso }
    res = 3 > 4 ; { mayor } { res se le asigna falso }
    res = 14 < 4 ; { menor } { res se le asigna falso }
    res = 14 <= 44 ; { menor igual } { res se le asigna verdadero }
    res = 0 >= -5 ; { mayor igual } { res se le asigna verdadero }

    res = 'Hola' = 'Hola' ; { igual } { res se le asigna verdadero }
    res = 'Hola' <> 'Alfredo' ; { distinto } { res se le asigna verdadero }

    res = verdadero = verdadero ; { igual } { res se le asigna verdadero }
    res = falso <> verdadero ; { distinto } { res se le asigna verdadero }
finproc ;
```

Los operadores de comparación mayor, menor, mayor igual y menor igual, solo pueden usarse en expresiones numéricas. Si uno quiere romper esta regla, y usarlos igualmente, el compilador GarGar le dirá que no está permitido usar los comparadores de mayor, menor, etc en expresiones del tipo texto o booleanas.

Veamos el siguiente ejemplo para ilustrar como son los tipos de cada expresión en la siguiente asignación:

```
res = ( 14 < 4 ) or verdadero ;
```

Comencemos por el menor. El < es un operador de comparación booleano entre números, así que espera que ambos operandos sean números. Sin embargo, < en si es un operador del tipo booleano, así que devuelve un booleano. Por otro lado, el **or** es un operador entre booleanos, así que el compilador GarGar esperará que ambos operandos sean del tipo booleano. En este caso, contamos con verdadero que es booleano, y con el resultado de (14 < 4), que como ya dijimos, es booleano.

Por último, existe un tipo de operador sumamente especial, ya que es un operador que se aplica a un solo termino. Este operador del que hablamos es el operador de negación (su símbolo es el !), que sirve para invertir un booleano (si era falso lo hace verdadero y viceversa). Siempre que se usa, hay que encerrar lo que se vaya a negar entre paréntesis:

```
procedimiento ejemplo5 ( )  
var x : booleano ;  
var z : booleano ;  
var res : booleano ;  
comenzar  
  res = ! ( verdadero ); { res se le asigna falso }  
  res = !(4 = 4) ; { res se le asigna falso }  
  res = !(3 > 4 ); { res se le asigna verdadero }  
  res = ! ( ! ( verdadero )); { res se le asigna verdadero }  
  
finproc ;
```

3.7 *¿Qué es un arreglo? (o como un arreglo se parece sospechosamente a un montón de variables juntas)*

Un arreglo es un conjunto de variables agrupadas con un mismo nombre. Al momento de declararlo, se define cuantas variables va a ocupar y luego se puede acceder a cada variable mediante su posición en el arreglo.

En GarGar los arreglos pueden ser de 1 a N posiciones, máximo que se especifica al momento de declararlo.

```
procedimiento ejemplo1 ( )  
var x : numero;  
var arr : arreglo [10] de numero;  
comenzar  
    x = 3;  
    arr[1] = 1 ;    {cada posición del arreglo es como si fuera una  
variable individual}  
    arr[2] = x - 1 ;  
    arr[x] = arr[2] ;    {se pueden usar variables como índices del  
arreglo}  
finproc ;
```

El ejemplo contiene varias cosas que ver:

1. Un arreglo se declara usando la palabra **arreglo** seguido de la cantidad de posiciones que queremos que tenga el arreglo entre corchetes, seguido de la palabra **de** y luego el tipo de dato (los mismos 3 disponibles para las variables normales).

2. Para acceder a la posición de un arreglo, se coloca el nombre del arreglo, y luego entre corchetes, la posición explícita, o la expresión que resulte en un valor numérico.

NOTA: Existe un error muy común, que el compilador GarGar no siempre sabe identificar (en realidad no es que no sabe, sino que no puede. No te confundas: el compilador GarGar es muy sabio), que es el intentar acceder a posiciones por fuera del rango definido (1 a N). En caso de ocurrir eso, saldrá un error al momento de ejecutar el programa, y no en el momento de compilarlo.

```
procedimiento ejemplo1 ( )  
var arr : arreglo [10] de numero;  
comenzar  
    x = 3;  
    arr[15] = 1 ;    {arrojara un error al momento de ejecutar}  
finproc ;
```

3. Una posición de un arreglo, se puede usar de la misma manera que una variable. Se le puede asignar valores explícitos, resultados de expresiones, o también pueden ser usados en expresiones.

3.8 Operaciones entre arreglos

Ok, quizás no tendríamos que haber creado un subtítulo con este nombre, ya que **NO** existen operaciones entre arreglos en el lenguaje GarGar. Así es: **NO** existen.

O quizás lo creamos, para dejar bien en claro y remarcado que no hay operaciones entre arreglos en su totalidad.

4. Constantes

4.1 *¿Qué es una constante? (o la historia de la variable que no se dejaba modificar)*

Una constante es básicamente una variable a la cual no se le puede cambiar el valor. Se le asigna una vez, al momento de declararla, y luego, nunca más se le modifica su valor a lo largo del programa. Por esa razón se las llama **constantas**, ya que su valor permanece constante a lo largo del programa.

4.2 *¿Y si no se pueden modificar, en que se parecen a las variables? (o como las constantes se copian de los tipos de las variables)*

Cada constante, como una variable, se declara con un tipo de dato (los mismos 3 disponibles para las variables: número, texto y booleano). La diferencia en el momento de declararla, es que a la constante se le asigna un valor, que debe coincidir con el tipo de dato elegido. Aquí unos ejemplos:

constantas

```
const constanteNum : numero = 123;  
const nombreLenguaje : texto = 'GarGar';  
const constanteBool : booleano = verdadero;
```

¿Hay algo raro en este ejemplo verdad? Seguro entendés porque cada declaración de constantes empieza con **const**, ya que cada variable se declaraba usando la palabra reservada **var**. Lo que sí es nuevo es esa

palabra **constantes** al principio. Eso lo vamos a ver en detalle más adelante, pero ahora vamos a explicarlo muy brevemente:

Las constantes se declaran a nivel global en un espacio del programa especialmente destinado para las constantes, que se identifica con la palabra **constantes**. ¿Qué quiero decir a nivel global? Que no importa en qué procedimiento me encuentre, yo voy a poder usar esa constante. Esto puede ser shockeante y sumamente confuso ahora, pero prometo que más adelante será más claro.

4.3 ¿Y para que me sirven las constantes?

Sirven para todo lo mismo que sirven las variables, con la excepción de que no se les puede asignar nuevos valores. Todas las operaciones que se pueden realizar con variables, también se pueden realizar con constantes, y también corren todas las validaciones del tipo de dato usado en los operandos.

Otro uso bastante frecuente es para definir el rango máximo de un arreglo. De esta manera se define el valor del rango máximo como constante, y en caso de necesitar cambiar la cantidad de posiciones del arreglo más tarde, basta con modificar el valor de la misma.

```
constantes
const constante : numero = 123;

procedimiento ejemplo1 ( )
var x : numero;
var arr : arreglo [constante] de numero;
comenzar
    x = 3;
    arr[1] = 1 ;
    arr[2] = x - 1 ;
    arr[x] = 3 ;
finproc ;
```

4.4 *¿Y qué pasa si yo quiero asignarle un nuevo valor a una constante?*

Simple, en ese caso, el compilador de GarGar te mostrará un mensaje de error informando que las constantes no pueden ser modificadas. Al compilador GarGar no le gusta que se rompan las reglas, así que no hará excepciones. Si tenés la necesidad de modificar una constante, es porque en realidad no deberías haberla declarado como constante en un principio. Declárala como variable, asigne el valor inicial en otro lado, y problema resuelto.

5. Estructura de un programa GarGar

5.1 *Espacio de declaraciones globales (hogar de las variables globales y las constantes)*

¡Otra vez eso de las declaraciones globales! Tranquilidad, que es mucho más simple de lo que suena con todas esas palabras que se le agregan.

Vamos a empezar por las variables: una variable puede ser global o local. ¿Qué me marca esto? En que una variable local, solo se puede usar dentro del procedimiento en la cual fue declarada, mientras que una variable global puede ser usada en cualquier procedimiento del programa. Miremos este ejemplo:

```
procedimiento SALIDA ( )  
  var x : numero ;  
  comenzar  
    x = 4 ; {esta asignación se realizara correctamente}  
  finproc ;  
  
procedimiento principal ( )  
  comenzar  
    x = 23 ; {esto arrojará error que la variable no está declarada}  
    llamar SALIDA ( ) ;  
  finproc ;
```

En este caso tenemos los dos procedimientos obligatorios del lenguaje GarGar y en salida declaramos la variable x de tipo número. La variable x es local al procedimiento salida, así que solo puede ser usada allí.

Por el contrario, miremos este ejemplo:

```
variables  
var global : numero;  
  
procedimiento ejemplo5()  
comenzar  
    global = 13; {esta asignación se realizara correctamente}  
finproc;  
  
procedimiento principal ( )  
comenzar  
    global = 23; {esto arrojara error que la variable no está declarada}  
    llamar SALIDA ( ) ;  
finproc
```

Fíjense que antes de la declaración de los procedimientos, tenemos la palabra **variables** y debajo de ella, la declaración de la variable global. Todas las variables que se declaren debajo de la palabra **variables** serán variables globales.

Un dato para los curiosos: no fue casualidad que no usáramos el procedimiento salida como hicimos en el ejemplo anterior; el procedimiento salida no admite que se utilicen variables globales dentro de él. Esto es una restricción del lenguaje GarGar, que el compilador no permite que se viole.

5.2 Variables globales vs Variables locales (¿Quién es más importante?)

Existe un tema de importancia entre las variables: Las variables locales siempre vienen primero. ¿Pero a que nos referimos? Queda más claro en este ejemplo:

```
variables
var x : numero;
var y : numero;

procedimiento ejemplo5()
comenzar
    x = 30; {Aquí estoy modificando la variable global x}
    y = 13; {Aquí estoy modificando la variable global y}
finproc;

procedimiento principal ( )
var x : numero;
comenzar
    x = 10; {Aquí estoy modificando la variable local x}
    y = 23; {Aquí estoy modificando la variable global y}
    llamar SALIDA ( ) ;
finproc ;
```

¡Fíjense que en este caso, existe una variable global con el mismo nombre que una variable local (la del procedimiento Principal)! Sin embargo, el compilador GarGar tiene bien claro cómo manejar estos casos: Cuando ve que se está usando una variable dentro de un procedimiento, lo primero que hace es fijarse si el nombre de esa variable ya está declarado en el procedimiento en el que me encuentro. Si está declarada, el compilador

GarGar usa esa. Si no estuviera declarada, el compilador se fija si la variable está declarada en el espacio de variables globales. Si esta, usa esa, y si no, arroja un error que la variable no está declarada.

¿Vieron que no era tan difícil?

5.3 *¿Existen también los arreglos como variable global?*

Así es. Todo lo explicado hasta el momento para variables globales, aplica a los arreglos.

5.4 *¿Y las constantes globales?*

¡Cierto! Nos estábamos olvidando de las constantes. Las constantes ya vimos algo antes. Volvamos a verlo:

constantes

```
const constanteNum : numero = 123;
```

```
const nombreLenguaje : texto = 'GarGar';
```

```
const constanteBool : booleano = verdadero;
```

Es como en las variables globales: debajo de la palabra **constantes** se definen todas las constantes.

En este caso, no hay problemas con las constantes locales, ya que **NO** existen las constantes locales. Todas las constantes se deben declarar en el espacio de declaraciones globales.

Ejemplo con constantes y variables globales:

constantes

const constanteNum : **numero** = 123;

const nombreLenguaje : **texto** = 'GarGar';

const constanteBool : **booleano** = **verdadero**;

variables

var x : **numero**;

var y : **numero**;

Es importante el orden, ya que primero se deben definir las constantes y luego las variables. Si uno quiere hacer al revés, el compilador GarGar le mostrará un error indicado que eso no es posible.

5.5 Espacio de declaraciones de procedimientos (¡el orden es importante!)

Es momento de contar un secreto: no solo se pueden declarar variables, constantes y procedimientos. También se pueden declarar lo que se llaman funciones.

Comparativamente, una casi igual a un procedimiento, con la excepción de que la función devuelve un valor, y no se usa la palabra **llamar** para usarlas. Más adelante hay un capítulo dedicado exclusivamente a las funciones, así que no nos detendremos en eso. Lo que era importante ahora, es saber que se pueden declarar tanto procedimientos y funciones.

El espacio de declaraciones de procedimientos y funciones tiene estas restricciones:

1. Si una procedimiento/función llama a otro, el llamado debe estar declarado antes. Ejemplo:

```
funcion ejemplo3 ( ) : numero
comenzar
finfunc 1231;

procedimiento ejemplo4 ( )
var x : numero ;
comenzar
    x = ejemplo3();
finproc;

procedimiento SALIDA ( )
comenzar
finproc ;

procedimiento principal ( )
comenzar
    llamar ejemplo4();
    llamar SALIDA ( ) ;
finproc ;
```

Este ejemplo es correcto, ya que están todos declarados en el orden correcto. Principal llama al procedimiento ejemplo4, y ejemplo4 usa la función ejemplo3.

En este otro caso, en cambio, podemos ver como ejemplo5 está declarado después de que ejemplo3 lo use, cosa que no está permitida en GarGar:

```
funcion ejemplo3 ( ) : numero
comenzar
  llamar ejemplo5(); {ejemplo 5 no esta declarado}
finfunc 1231;

procedimiento ejemplo5 ( )
comenzar
finproc;

procedimiento ejemplo4 ( )
var x : numero ;
comenzar
  x = ejemplo3();
finproc;

procedimiento SALIDA ( )
comenzar
finproc ;

procedimiento principal ( )
comenzar
  llamar ejemplo4();
  llamar SALIDA ( ) ;
finproc ;
```

2. Principal debe ser el último procedimiento declarado, y Salida el anteúltimo. Esto ya lo vimos anteriormente, y es un requerimiento del lenguaje GarGar.

Ejemplo de un programa entero:

```
constantes  
const c1 : texto = 'Buen dia' ;  
  
variables  
var xt : booleano ;  
  
funcion ejemplo3 ( ) : numero  
comenzar  
finfunc 1231 ;  
  
procedimiento ejemplo4 ( )  
var x : numero ;  
comenzar  
    x = ejemplo3 ( ) ;  
finproc ;  
  
procedimiento SALIDA ( )  
comenzar  
finproc ;  
  
procedimiento principal ( )  
comenzar  
    llamar ejemplo4 ( ) ;  
    llamar SALIDA ( ) ;  
finproc ;
```

6. Procedimientos

6.1 *¿Qué es un procedimiento? (¡Decime por favor, para que lo puedo usar!)*

Un procedimiento es una secuencia de sentencias que se agrupan dentro de un bloque, con el fin de poder ser utilizada más de una vez. Las sentencias generalmente se agrupan de acuerdo a cierta lógica que luego se ve reflejada en el nombre del procedimiento.

Miremos este ejemplo:

```
variables  
var x : numero ;  
var y : numero ;  
var z : numero ;  
  
procedimiento InicializarVariables ( )  
comenzar  
    x = 12;  
    y = 34;  
    z = 56;  
finproc ;
```

Este procedimiento lo que se encarga es de inicializar variables. La ventaja de poner esto en un procedimiento, es que cada vez que quiera inicializar las variables, simplemente tengo que llamar al procedimiento.

Otra ventaja, es que si yo tengo varios lugares en el código donde tengo que inicializar mis variables, y tengo que agregar una variable mas, de no estar usando un procedimiento, lo que yo tendría que hacer es modificar

el código en cada uno de los puntos. Por el contrario, si uso un procedimiento, lo tengo que modificar una sola vez y listo.

6.2 *¿Me gusta el nombre “proc” para mis procedimientos, lo puedo usar para todos?*

No, los nombres no se deben repetir. Entre procedimientos y funciones no debe haber ningún nombre repetido.

6.3 *¿Me recordás lo de las variables locales?*

¡Si cómo no! Los procedimientos pueden tener variables que solo existen dentro del espacio del procedimiento declarado.

Si estas variables, tienen el mismo nombre que alguna variable global, **SIEMPRE** se usara la variable local.

6.4 *Parámetros de un procedimiento (¿o más tipos de variables?)*

Los procedimientos tienen otra característica, que se llama parámetros. Un parámetro es muy parecido a una variable local, con la excepción que es una variable que se le puede pasar desde quien lo llama. Para decirlo de otra manera, un parámetro permite pasarle una variable, con su valor, desde afuera, y dentro del procedimiento se comporta como una variable local más.

¡OJO! Como se comporta como una variable mas, no podemos tener un parámetro que se llame de la misma manera que una variable local. El compilador GarGar es estricto en esto (también)

```
procedimiento SALIDA ( x : numero , y : numero, z : numero )
comenzar
    mostrar(x);
    mostrar(y);
finproc ;

procedimiento principal ( )
var x : numero ;
var y : numero ;
comenzar
    x = 10;
    y = 20;
    llamar SALIDA ( x, y , 10) ;
finproc ;
```

Mucha atención al anterior ejemplo, que tiene muchas cosas muy importantes. Volvemos al primer ejemplo del tutorial, el procedimiento Principal y Salida.

Lo que estamos haciendo en este caso es definir 3 parámetros en el procedimiento Salida. Fíjense como en **llamar**, después de colocar el nombre del procedimiento, entre paréntesis se colocan las variables o valores de los parámetros. En este ejemplo a modo ilustrativo, colocamos 2 variables como parámetro y el restante como un valor explícito.

A no olvidarse, que si una variable puede ser parámetro de un procedimiento, entonces un arreglo también puede serlo:

```
procedimiento ejemplo1 ( arr : arreglo [10] de numero )  
comenzar  
    arr[1] = 1 ;  
finproc ;
```

6.5 Tipos de parámetros (por dios, cuantos tipos hay)

Los parámetros también tienen 2 tipos. Están los que se llaman por valor, o los que se llaman por referencia. Se los llama de esta manera por cómo se comporta aquellos que se los coloca como parámetro.

Únicamente las variables pueden ser pasadas como parámetro por referencia. Los valores explícitos solo pueden ser pasados como parámetros por valor.

En el caso de un parámetro por valor, si es un valor explícito, es como si no ocurriera nada, simplemente se toma el valor. Si es una variable, tomo la variable y se realiza una copia. ¿Qué quiero decir con esto? Que una

vez terminado el procedimiento, no importa cómo se modifiko el valor del parámetro, la variable usada en el llamado, permanece con el mismo valor que antes de entrar al procedimiento.

En el caso de un parámetro por referencia, tomo la variable que se coloco como parámetro y uso **ESA MISMA**. O sea, que una vez terminado el procedimiento, la variable usada en el llamado tiene el valor con el que termino el procedimiento.

```
procedimiento ejemplo ( ref x : numero , y : numero )
comenzar
    x = 20 ;
    y = 30 ;
finproc ;

procedimiento SALIDA ( x : numero , y : numero )
comenzar
    mostrar ( x ) ;
    mostrar ( y ) ;
finproc ;

procedimiento principal ( )
var x : numero ;
var y : numero ;
comenzar
    x = 10 ;
    y = 50 ;
    llamar ejemplo ( 10, 10); {esto no compila porque no puedo pasar un
valor explicito a un parámetro definido por referencia}
```

```
llamar ejemplo ( x , y ) ; {x al ser por referencia ahora vale 20. y en  
cambio, al ser por valor, sigue valiendo 50}
```

```
llamar SALIDA ( x , y ) ;  
finproc ;
```

El ejemplo y sus comentarios se explican por sí mismos.

6.6 ¿Cómo uso un procedimiento?

¡Es muy simple! Hay que usar la palabra **llamar** seguido del nombre del procedimiento, y luego (entre paréntesis), pasar los parámetros que correspondan al tipo de datos definidos en el momento de la declaración.

6.7 El restrictivo procedimiento SALIDA

¿Había dicho que no iba a poner más en mayúsculas el procedimiento SALIDA no? Bueno, mentí.

El procedimiento salida es muy particular, ya que tiene restricciones sobre que puede contener. Esto no es un capricho, sino que es para formar en una práctica correcta de separar los resultados del cálculo de los mismos.

Estas son las restricciones del procedimiento salida:

- No se permite el uso de variables globales dentro de el.
- No se permite modificar los parámetros del mismo.
- No se permite asignar los parámetros a variables locales del procedimiento salida.
- No se permite el uso de la sentencia **leer**.
- No se permite el uso de la sentencia **llamar**.

7. Funciones

7.1 *¿Qué es una función? (se ve como un procedimiento con un tipo de dato)*

Decir que una función se ve parecido a un procedimiento no es de todo cierto. Se ven muy parecidos en su declaración:

```
funcion ejemplo3 ( x : numero , y : numero , z : numero ) : numero
comenzar
finfunc 1231 ;

procedimiento ejemplo4 ( x : numero , y : numero , z : numero )
var x : numero ;
comenzar
    x = ejemplo3 ( ) ;
finproc ;
```

Pero en realidad son bastante distintos. Una función en sí misma, se usa para devolver un valor. O sea que el objetivo de una función es realizar una serie de pasos que terminan por devolver un valor a quien la llamo. En el caso del procedimiento, no se devuelve ningún valor.

La manera de usar una y otra es bien distinta:

```
funcion ejemplo3 ( ) : numero
comenzar
finfunc 1231 ;
```

```
procedimiento ejemplo4 ()  
comenzar  
finproc ;  
  
procedimiento SALIDA ( )  
comenzar  
finproc ;  
  
procedimiento principal ( )  
var x : numero ;  
comenzar  
    x = ejemplo3() + 200; {x va a ser 1431, que es lo que devuelve la  
función + 200}  
    llamar ejemplo4 ( ) ;  
    llamar SALIDA ( ) ;  
finproc;
```

Fíjense como la función se usa como si fuera una variable o un valor explícito y puede ser parte de una expresión como en el ejemplo.

Por último, las funciones se declaran con la palabra **funcion** y se terminan con **finfunc**, mientras que los procedimientos con la palabra **procedimiento** y se terminan con **finproc**.

7.2 *¡Quiero declarar una función! (explicame de una vez como hago)*

Bueno, a no desesperarse que esto es justamente lo más parecido a los procedimientos:

1. Uno debe elegir un nombre que sea representativo a lo que sea que haga la función.
2. Uno debe definir el tipo de parámetros y la cantidad que necesita para lo que se necesite programar.
3. Uno debe definir las variables locales de la función, que no deben repetirse con los nombres de los parámetros. Recordar que de existir una variable global con el mismo nombre que una variable local, el compilador GarGar siempre usara la variable local.
4. **Uno debe definir cuál es el tipo de dato del valor que retorna la función.**
5. **Uno debe definir como se arma el valor que retorna la función.**

Como pueden ver, salvo los puntos 4 y 5 que explicaremos a continuación, el resto son exactamente iguales a un procedimiento. Veamos la función Duplicar para explicar los puntos restantes:

```
funcion Duplicar ( x :numero ) : numero  
comenzar  
finfunc x * 2;
```

El tipo de dato que retorna la función es especificado después de los parámetros, como si estuviésemos definiendo una variable (el : y luego el tipo de dato).

Por otro lado, el valor que retorna la función se coloca después del **finfunc** y antes del ; que lo concluye. En el retorno, uno puede usar

cualquier expresión, simple o compleja, corta o extensa, siempre y cuando sea del mismo tipo que el definido en la declaración de la función. De no serlo, el compilador GarGar lo hará saber con un hermoso error.

7.3 ¿Que bueno, no veo la hora de declarar mi función que retorne un arreglo!

Mmmm, lamentablemente, eso no es posible. Una función únicamente devuelve un valor solo, no un arreglo.

7.4 ¿Y para que me sirve una función?

Una función sirve sobre todo, en casos donde se requiere un valor que requiere algún tipo de cálculo complejo, y que depende de datos de entrada. Colocando esto en una función, ganamos la ventaja de que puedo usar ese cálculo complejo en varios lugares del programa sin necesidad de copiar y pegarlo, y se gana la ventaja de que en caso de necesitar modificar el cálculo, la modificación se hace en un solo lugar.

8. Sentencias del lenguaje GarGar

En este capítulo vamos a ver aquellas sentencias que se pueden usar dentro de los procedimientos o funciones, dentro de lo que es el **comenzar** y el **finproc/finfunc**.

Las sentencias posibles son 6 y cada una tiene su propósito marcado. GarGar está pensado para que justamente ninguna sentencia repita su función con otra, justamente para no confundir al usuario.

Todas las sentencias que veremos a continuación pueden ser usadas en cualquier orden, e incluso veremos que hay 2 sentencias que permiten que se coloquen otras sentencias dentro de ellas.

Existen únicamente dos restricciones respecto a las sentencias que se pueden usar en los procedimientos:

1. No se puede usar la sentencia llamar dentro del procedimiento salida, ya que no se permite llamar a otros procedimientos dentro de él.
2. No se permite usar la sentencia leer dentro del procedimiento salida tampoco.

Salvadas estas dos restricciones, pasaremos a ver las sentencias del lenguaje GarGar.

8.1 ASIGNACION: La sentencia para colocarle valor a las variables.

Esta sentencia ya ha sido vista a lo largo del tutorial. Cumple con el fin de colocar el resultado de la expresión del lado derecho, en el lado izquierdo. Esta sentencia debe ser terminada con un ;

```
ASIGNABLE = EXPRESION ;
```

EXPRESION puede ser cualquier tipo de expresión con operadores, valores explícitos, variables y/o funciones.

Restricciones:

- ASIGNABLE debe ser una variable o una posición de un arreglo.
- El tipo resultante de la EXPRESION debe ser del mismo tipo que ASIGNABLE.
- EXPRESION puede ser valores explícitos, arreglos, variables, funciones, expresiones, etc (o sea, todo aquello que tenga un valor asociado). En caso de usar variables o funciones, éstas deben haber sido declaradas, y en el caso de función, sus parámetros estar correctos.

Ejemplos:

```
x = Duplicar ( 2 );  
y = 43;  
arr[1] = Duplicar (3) + 43 / 3;
```


8.2 LLAMAR: La sentencia para llamar a un procedimiento.

Esta es otra de las sentencias que se han visto en el tutorial. Se usa para indicar que se quiere llamar a un procedimiento. La cantidad de parámetros a colocar, dependerá pura y exclusivamente del procedimiento a llamar. Esta sentencia debe ser terminada con un ;

```
Llamar NOMBREPROC ( PARAMETROS ) ;
```

Restricciones:

- NOMBREPROC debe ser el nombre de un procedimiento ya declarado.
- La cantidad y tipos de parámetro de PARAMETROS debe coincidir con la cantidad y tipo declarados para NOMBREPROC.
- PARAMETROS puede ser valores explícitos, arreglos, variables, funciones, expresiones, etc (o sea, todo aquello que tenga un valor asociado). En caso de usar variables o funciones, éstas deben haber sido declaradas, y en el caso de función, sus parámetros estar correctos.

Ejemplos:

```
llamar SALIDA ( ) ;  
llamar ejemplo1 ( 12, 23, 'Hola', verdadero );  
llamar ejemplo2 ( Duplicar ( 2 ) + 4 , arr[1], 'Hola' & 'Adios',  
resultado and (arr[x + 1] > 30);
```

8.3 *MOSTRAR: La sentencia de impresión en pantalla*

Esta sentencia sirve para mostrar todo lo que se le pase por parámetro en pantalla. El número de parámetros es variable, se puede pasar 1 a N expresiones, y el sistema mostrara en pantalla el resultado de cada uno. Esta sentencia debe ser terminada con un ;

Mostrar (PARAMETROS) ;

NOTA: Existe una variante de Mostrar que es **MostrarP**, que además de mostrar por pantalla, lo que hace es esperar una entrada de teclado. Sirve para cuando se quiere mostrar con una pausa.

Restricciones:

- PARAMETROS debe contener al menos un parámetro. No se puede usar la sentencia Mostrar vacía.
- PARAMETROS puede ser valores explícitos, arreglos, variables, funciones, expresiones, etc (o sea, todo aquello que tenga un valor asociado). En caso de usar variables o funciones, estás deben haber sido declaradas, y en el caso de función, sus parámetros estar correctos.

Ejemplos:

```
mostrar ( ' El resultado final es: ', arr[1]);  
mostrar ( ' Los primeros 3 números son: ', 1, Duplicar(1), 3 );  
mostrarP ( ' El resultado final fue ', verdadero, ' y el número máximo  
fue: ', Duplicar(arr[1]));
```

8.4 LEER: La sentencia de lectura de datos por teclado

Esta sentencia sirve para tomar datos de entrada por teclado y asignárselos a una variable o posición de un arreglo. Permite que un usuario pueda personalizar la entrada de un programa al ejecutarlo. Esta sentencia debe ser terminada con un ;

```
Leer ASIGNABLE ;
```

Restricciones:

- ASIGNABLE debe ser una variable o una posición de un arreglo.
- ASIGNABLE debe ser del tipo **numero** o **texto**. No se admiten asignar valores a variables booleanas. En caso de necesitarlo, use los otros tipos para luego en base al valor, definir si es verdadero o falso.
- En caso de asignar un valor no numérico a una variable del tipo **numero**, el programa arrojará un error en ejecución.

Ejemplos:

```
leer num;  
leer cadena;  
leer arr[x + 4];
```

8.5 *MIENTRAS: La estructura iteradora (iteradora, iteradora, iteradora...)*

Esta sentencia es diferente a las vistas hasta ahora. Sobre todo porque no se la llama sentencia, sino estructura; una estructura iteradora. ¿Qué queremos decir con esta palabra compleja? Queremos decir que es una estructura que permite que se repitan otras sentencias, siempre que se cumpla una condición.

```
mientras ( CONDICION ) hacer
    BLOQUE
finmientras;
```

En lenguaje informal, la lectura de esta estructura sería “mientras se cumpla la condición booleana especificada entre los paréntesis, voy a repetir las sentencias (en orden) que tenga entre el **hacer** y el **finmientras**”.

Restricciones:

- CONDICION puede ser una expresión o un valor explícito, pero debe terminar siendo del tipo **booleano**.
- El compilador GarGar no hace un chequeo sobre si la condición del mientras alguna vez se cumplirá. En caso de realizar una iteración que no concluye, en la ejecución del programa aparecerá un error de iteración infinita.
- BLOQUE puede ser de 0 a N sentencias de las especificadas en este capítulo, incluidas la estructura condicional y la estructura iteradora.

Ejemplos:

```
mientras (x < 10) hacer
```

```
    mostrar(x);
```

```
    x = x+1;
```

```
finmientras;
```

```
mientras ((Duplicar (y) + 43 / 3) < Duplicar (133)) hacer
```

```
    mientras (x < 10) hacer
```

```
        mostrar(x);
```

```
        x = x+1;
```

```
    finmientras;
```

```
    y = y+1;
```

```
finmientras;
```

8.6 SI: La estructura condicional (¿es esta o la otra?)

Esta sentencia es del mismo tipo que el **mientras**. No es una sentencia, sino una estructura; una estructura condicional. O sea, lo que queremos decir que es una estructura que elige ejecutar unas sentencias u otras, dependiendo de una condición.

```
si ( CONDICION ) entonces
    BLOQUE
finsi;
```

En lenguaje informal, la lectura de esta estructura seria “si se cumple la condición, voy a ejecutar las sentencias en el BLOQUE_VERDADERO”.

OJO: Esta estructura tiene una parte opcional que es un bloque de sentencias para ejecutar en el caso que la condición resulte falsa:

```
si ( CONDICION ) entonces
    BLOQUE_VERDADERO
sino
    BLOQUE_FALSO
finsi;
```

En lenguaje informal, la lectura de esta estructura seria “si se cumple la condición, voy a ejecutar las sentencias en el BLOQUE_VERDADERO; sino, voy a ejecutar las sentencias en el BLOQUE_FALSO”.

Restricciones:

- **CONDICION** puede ser una expresión o un valor explícito, pero debe terminar siendo del tipo **booleano**.
- **BLOQUE** puede ser de 0 a N sentencias de las especificadas en este capítulo, incluidas la estructura condicional y la estructura iteradora.

Ejemplos:

```
si (x < 10) entonces
    Mostrar('Es menor a 10');
finsi;
```

```
si (x = verdadero) entonces
    Mostrar('Es verdadero');
sino
    Mostrar('Es falso');
finsi;
```


9. Ejemplos de programas completos

9.1 *Ejemplo 1: Algoritmo de Ordenamiento por Burbujeo*

CONSTANTES

CONST TOPE : **numero** = 10;

VARIABLES

VAR arr : **ARREGLO**[TOPE] **de** **numero**;

PROCEDIMIENTO Burbujeo()

VAR i,j,tmp : **numero**;

COMENZAR

i = TOPE - 1;

MIENTRAS (i >= 1) **HACER**

 j = 1;

MIENTRAS (j <= i) **HACER**

SI (arr[j] > arr[j + 1]) **ENTONCES**

 tmp = arr[j];

 arr[j] = arr[j + 1];

 arr[j + 1] = tmp;

FINSI;

 j = j + 1;

FINMIENTRAS;

 i = i - 1;

FINMIENTRAS;

FINPROC;

PROCEDIMIENTO SALIDA()**COMENZAR**

{Colocar acá los elementos que quiera mostrar}

FINPROC;**PROCEDIMIENTO PRINCIPAL()**

VAR i : **numero**;

COMENZAR

arr[1] = 9;

arr[2] = 8;

arr[3] = 67;

arr[4] = 43;

arr[5] = 21;

arr[6] = 90;

arr[7] = 2;

arr[8] = 54;

arr[9] = 20;

arr[10] = 4;

LLAMAR Burbujeo();

i = 1;

MIENTRAS (i <= TOPE) **HACER**

MOSTRAR(arr[i]);

 i = i + 1;

FINMIENTRAS;

LLAMAR SALIDA();

FINPROC;

9.2 *Ejemplo 2: Algoritmo de Raíz cuadrada por aproximación*

constantes

const MaxArreglo : **numero** = 10;

variables

var ty : **booleano**;

funcion RaizAprox(num : **numero**) : **numero**

var N, i, U, aux, aux2: **numero**;

comenzar

N = 100;

U = 1;

i = 1;

mientras (i <= N) **hacer**

 aux = U+num;

 aux2 = U+1;

 U = aux/aux2;

 i = i + 1;

finmientras;

finfunc U;

procedimiento Salida(num : **numero**)

comenzar

 mostrar(num);

finproc;

procedimiento PRINCIPAL()

var num : **numero**;

comenzar

MOSTRAR ('Ingrese un numero para calcular su raíz cuadrada:');

LEER num;

llamar Salida(RaizAprox(num));

finproc;

10. Funciones ya provistas por el lenguaje GarGar

El lenguaje GarGar ya trae incorporadas funciones de uso frecuente. A continuación las listaremos:

10.1 *Función EsImpar*

Tipo: **booleano**

Descripción: Función que chequea si la parte entera de un número es impar.

Parámetros

1. num : **numero**

Número con o sin decimales a chequear si es impar.

Ejemplo

```
x = ESIMPAR(3); {Esto da verdadero}
x = ESIMPAR(4.15); {Esto da falso}
x = ESIMPAR(17.14); {Esto da verdadero}
```

10.2 Función EsPar

Tipo: **booleano**

Descripción: Función que chequea si la parte entera de un número es par.

Parámetros

1. num : **numero**

Número con o sin decimales a chequear si es par.

Ejemplo

```
x = ESPAR(2); {Esto da verdadero}  
x = ESPAR(3.14); {Esto da falso}  
x = ESPAR(4.14); {Esto da verdadero}
```

10.3 *Función Potencia*

Tipo: **numero**

Descripción: Función que permite realizar cálculos de potencia sobre una base y un exponente que se pasan por parámetro.

Parámetros

1. base : **numero**
Número base sobre el cual se hace la potencia.
2. exp : **numero**
Número exponente el que se eleva la base.

Ejemplo

```
x = POTENCIA(2, 8 ); {Esto da 64}  
x = POTENCIA(4, 2 ); {Esto da 16}  
x = POTENCIA(10, 3 ); {Esto da 1000}
```

10.4 Función Raíz

Tipo: **numero**

Descripción: Función que permite realizar cálculos de potencia sobre una base y un exponente que se pasan por parámetro.

Parámetros

1. base : **numero**
Número base sobre el cual se hace la raíz.
2. raiz : **numero**
Número exponente sobre el que se realiza la raíz.

Ejemplo

```
x = RAIZ(9, 2 ); {Esto da 3}
```

```
x = RAIZ(-9, 2 ); {Esto arroja error. No se pueden calcular raíces con  
base negativa}
```

```
x = RAIZ(9, -2 ); {Esto arroja error. No se pueden calcular raíces con  
exponente negativo}
```


10.5 Función Redondear

Tipo: **numero**

Descripción: Función que redondea a entero el número pasado por parámetro.

Parámetros

1. num : **numero**

Número con o sin decimales a redondear.

Ejemplo

```
x = REDONDEAR(3); {Esto devuelve 3}  
x = REDONDEAR(4.15); {Esto devuelve 4}  
x = REDONDEAR(17.74); {Esto devuelve 18}
```

10.6 *Función Truncar*

Tipo: **numero**

Descripción: Función que trunca la parte decimal de un número, si esta tuviere.

Parámetros

1. num : **numero**
Número con o sin decimales a truncar.

Ejemplo

```
x = TRUNCAR(3); {Esto devuelve 3}  
x = TRUNCAR(4.15); {Esto devuelve 4}  
x = TRUNCAR(17.14); {Esto devuelve 17}
```

10.7 Función ValAbs

Tipo: **numero**

Descripción: Función que devuelve el valor absoluto de un número.

Parámetros

1. num : **numero**

Número con o sin decimales a calcular el valor absoluto.

Ejemplo

```
x = VALABS(3); {Esto devuelve 3}  
x = VALABS(-4.15); {Esto devuelve 4.15}  
x = VALABS(-17.14); {Esto devuelve 17.14}
```

10.8 *Función Pi*

Tipo: **numero**

Descripción: Función que devuelve el número PI.

Parámetros

No tiene

Ejemplo

```
x = PI(); {Esto devuelve 3.1415926535}
```

10.9 Función Seno

Tipo: **numero**

Descripción: Función que calcula el seno en grados.

Parámetros

1. num : **numero**
Número en grados a calcular el seno.

Ejemplo

```
x = SENO(0); {Esto devuelve 0}  
x = SENO(45); {Esto devuelve 0.7071067}  
x = SENO(90); {Esto devuelve 1}
```

10.10 Función rSeno

Tipo: **numero**

Descripción: Función que calcula el seno en grados.

Parámetros

1. num : **numero**
Número en radianes a calcular el seno.

Ejemplo

```
x = RSENO(0); {Esto devuelve 0}  
x = RSENO(PI()/4); {Esto devuelve 0.7071067}  
x = RSENO(PI()/2); {Esto devuelve 1}
```

10.11 Función Coseno

Tipo: **numero**

Descripción: Función que calcula el coseno en grados.

Parámetros

1. num : **numero**

Número en grados a calcular el coseno.

Ejemplo

```
x = COSENO(0); {Esto devuelve 1}  
x = COSENO(45); {Esto devuelve 0.7071067}  
x = COSENO(90); {Esto devuelve 0}
```

10.12 Función rCoseno

Tipo: **numero**

Descripción: Función que calcula el coseno en radianes.

Parámetros

1. num : **numero**

Número en radianes a calcular el coseno.

Ejemplo

```
x = RCOSENO(0); {Esto devuelve 1}  
x = RCOSENO(PI()/4); {Esto devuelve 0.7071067}  
x = RCOSENO(PI()/2); {Esto devuelve 0}
```

10.13 Función Tangente

Tipo: **numero**

Descripción: Función que calcula la tangente en grados.

Parámetros

1. num : **numero**

Número en grados a calcular la tangente.

Ejemplo

```
x = TANGENTE(0); {Esto devuelve 0}  
x = TANGENTE(45); {Esto devuelve 1}
```

10.14 Función rTangente

Tipo: **numero**

Descripción: Función que calcula la tangente en radianes.

Parámetros

1. num : **numero**

Número en radianes a calcular la tangente.

Ejemplo

```
x = RTANGENTE(0); {Esto devuelve 0}  
x = RTANGENTE(PI()/4); {Esto devuelve 1}
```


11. Buenas prácticas del lenguaje GarGar

En este capítulo enumeraremos algunas de las mejores prácticas al programar en el lenguaje GarGar, y otras prácticas que aplican a cualquier lenguaje de programación. Son pocas porque queremos que se te graben rápidamente en la cabeza.

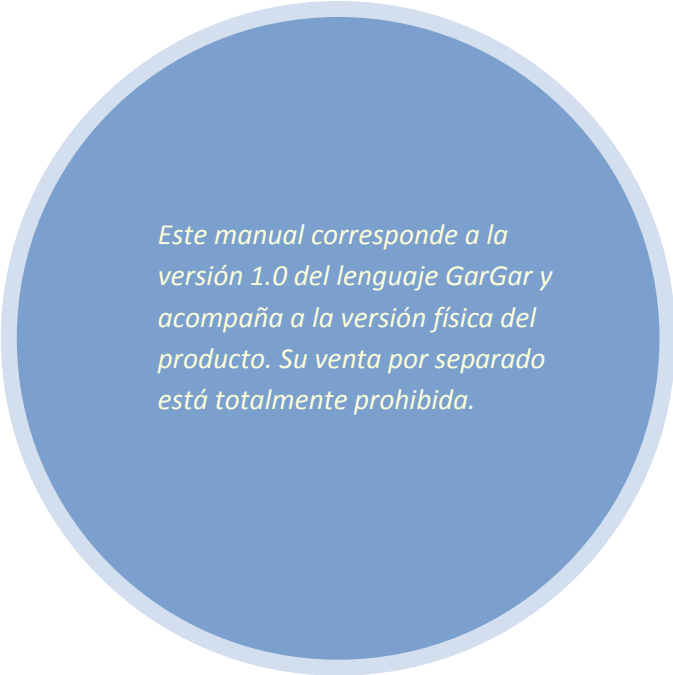
1. No tengas miedo de colocar nombres largos y descriptivos a tus variables y parámetros. De esa manera, sabes inmediatamente que debe tener la variable.

```
var x : numero;  
var numIntentosInvalidos : numero;
```

¿Cuál de las 2 variables se entiende mejor que hace?

2. Usa una convención de nombres para tus para tu código. De esta manera puedes identificar rápidamente:
 - a. Las variables empiezan en minúscula, y cada nueva palabra en el nombre, en mayúscula.
Ejemplo: numIntentosInvalidos.
 - b. Las funciones y procedimientos empiezan con mayúscula, y cada nueva palabra en el nombre, en mayúscula.
Ejemplo: DuplicarSiEsImpar
3. Ningún procedimiento/función debería ser más grande que 20 líneas en su cuerpo. Siempre que puedas, intenta agrupar lógicamente pedazos de tu programa, con el fin de agruparlos en procedimientos. De esta manera, tendrás muchos procedimientos cortos y legibles.

-
4. Si tenés que copiar dos veces el mismo código en distintas partes del programa, estás haciendo algo mal. Intenta poner ese código en una función/procedimiento, y si no son idénticos, usa un parámetro para diferenciarlos al momento de llamarlos.
 5. Intenta evitar el uso de variables globales, a no ser que sean estrictamente necesarias.
 6. Identá tu código y mantenelo ordenado. Es la mejor manera de que vos y otra persona entiendan que es lo que hace el programa.
 7. No abuses de los comentarios, pero que tampoco falten. Los comentarios deben existir en todos los casos en donde el código no se explica por sí solo.
 8. Siempre que pongas un comentario, colocale la fecha en el que lo escribiste.



Este manual corresponde a la versión 1.0 del lenguaje GarGar y acompaña a la versión física del producto. Su venta por separado está totalmente prohibida.