

# Contadores y acumuladores

Sitio:

[Agencia de Aprendizaje a lo largo de la Vida](#)

Curso:

Tecnicas de Programación 1° G

Libro:

Contadores y acumuladores

Imprimido por:

Eduardo Manuel Moreno

Día:

sábado, 7 de septiembre de 2024, 01:11

# Tabla de contenidos

- 1. Contadores
- 2. Acumuladores
- 3. Seguimiento y depuración de algoritmos
  - 3.1. Prueba y depuración
  - 3.2. Tipos de errores
- 4. Documentación
- 5. Prueba de escritorio para validar algoritmos.
  - 5.1. Nuestro algoritmo

# 1. Contadores

Un contador, no es más que una variable que cuenta (¿parece obvio, no?).

## Veamos un ejemplo de la vida cotidiana

Al ir de compras a una farmacia, los clientes para obtener un turno deben tomar un ticket. Un letrero electrónico indica el número del cliente que se está atendiendo, luego éste número cambia incrementándose en 1 para anunciar el siguiente turno a ser atendido.



El ejemplo, es un uso práctico de un contador, con el que se observan ciertas características:

- Siempre tienen un valor inicial.
- Su valor nuevo es el resultado del valor anterior más una constante (en nuestro ejemplo: 1).

Al inicio del día, el **contador debe ser inicializado**, de preferencia con 0, cuando un vendedor está listo para atender a un cliente, el contador se incrementa en uno, se escucha una alerta y se puede acercar el cliente con el primer turno.

Las características descritas para forma algorítmica se escriben como:

```
contador ← 0
```

```
contador ← contador + 1
```

Se puede leer como: contador recibe el resultado del valor que tenía sumado el valor de 1 o incrementando en 1.

Desde luego que a los contadores pueden sumarse un valor diferente a 1, pero siempre será un valor constante; también pueden tener cambios de forma ascendente, o disminuir desde un valor inicial.



Un ejemplo de contador decreciente se observa en el cronómetro del microondas para calentar alimentos.

- El **valor inicial** son los segundos que permanecerá encendido.

- El contador de tiempo disminuye en uno cada segundo y al llegar a 0 se apaga el microondas.

Las características descritas para forma algorítmica se escriben como:

```
cronometro ← 45
```

```
cronometro ← cronometro - 1
```

En el siguiente ejemplo,

- la variable que oficia de contador es la variable “par”, normalmente son inicializadas en 0 antes de un ciclo y suelen aumentar su valor ( $par = par + 1$ ) si se cumplen ciertas condiciones, en este caso, cuenta la cantidad de números pares que hay entre 1 y 10.

```
Algoritmo cantidadPares
Definir cantidadNumerosPares, i Como Entero
cantidadNumerosPares = 0
Para i=1 Hasta 10
    Si (i%2==0)
        cantidadNumerosPares = cantidadNumerosPares + 1
    FinSi
FinPara
Escribir "La cantidad de numeros pares es: ", cantidadNumerosPares
FinAlgoritmo
```

## 2. Acumuladores

Un **acumulador** en programación es una versión ampliada de un contador y tiene las mismas características que un contador excepto el valor de incremento que es un valor variable



### Veamos un ejemplo de la vida cotidiana

Una cuenta de ahorros puede representarse en un algoritmo mediante un acumulador, pues el ahorrista no siempre podrá ahorrar una cantidad fija en la cuenta, un día deposita 10, otro día deposita 30, otro deposita 5.

Con el ejemplo de ahorro, se puede determinar que:

- en el **acumulador** no siempre se añade un valor positivo, pues cuando se hace un retiro, se puede interpretar como que el valor añadido es negativo.

Las características descritas para forma algorítmica se escriben como:

```
acumulador ← 0
```

```
acumulador ← acumulador + X
```

La expresión del literal para una cuenta puede leer como: «saldo nuevo» de acumulador es el «saldo anterior» de acumulador considerando el deposito (+x) o retiro (-x).

Recordar que:



El concepto de asignar es usado en algoritmos “=” carece de sentido matemático.

Si tomamos como ejemplo el acumulador, si se expresa como una igualdad, se interpretaría como:

```
acumulador = acumulador + X
```

```
acumulador = acumulador - X
```

Cuando en el algoritmo se quiere expresar en realidad es una asignación, por lo que se utiliza el símbolo “←” o “=”

```
acumulador ← acumulador + X
```

Esta aclaración permite formalizar la diferencia de comparación de igualdad “==” usada dentro de los condicionales con el de asignación de un valor mediante “=”.

Ejemplo:



### Veamos otros ejemplo.

Pedir al usuario notas de alumnos. Este ingreso de notas finalizará cuando la nota tenga el valor -1. Se pide obtener el promedio de todas las notas ingresadas.

```

Algoritmo promedioNotas
  Definir notaIngresada, contadorNotas, acumuladorNotas Como Entero
  Definir promedio Como Real
  // Le damos un valor inicial a las variables que oficiarán de
  // contador y de acumulador
  contadorNotas = 0
  acumuladorNotas = 0
  //Pedimos la primer nota en una lectura anticipada al ciclo
  Escribir "Ingresa una nota: "
  Leer notaIngresada
  Mientras (notaIngresada ≠ -1) Hacer
    //Contamos 1 nota ingresada
    contadorNotas = contadorNotas + 1
    //Acumulamos la nota ingresada
    acumuladorNotas = acumuladorNotas + notaIngresada
    //Pedimos la siguiente nota
    Escribir "Ingresa una nota: "
    Leer notaIngresada
  FinMientras
  //Cuando ingresan -1 salimos del ciclo y chequeamos
  //si se ingreso alguna nota así no dividimos por cero
  Si (contadorNotas ≠ 0)
    promedio = acumuladorNotas / contadorNotas
    Escribir "El promedio de las notas ingresadas es: ", promedio
  SiNo
    Escribir "No se han ingresado notas para procesar."
  FinSi
FinAlgoritmo

```

### 3. Seguimiento y depuración de algoritmos



La **codificación** es la operación de escribir la solución del problema (de acuerdo a la lógica del diagrama de flujo o pseudocódigo), en una serie de instrucciones detalladas en un código reconocible por la computadora, la serie de instrucciones detalladas se le conoce como programa fuente, el cual se escribe en un lenguaje de programación o lenguaje alto nivel.



## 3.1. Prueba y depuración

Los errores dentro de la programación de computadoras son muchos y aumentan considerablemente con la complejidad del problema.

Y por lo tanto, el proceso de prueba permite identificar y eliminar errores (bugs), para dar paso a una solución sin errores se le llama depuración.



La depuración o prueba resulta una tarea tan creativa como el mismo desarrollo de la solución, por ello se debe considerar con el mismo interés y entusiasmo.

Resulta conveniente observar los siguientes principios al realizar una depuración, ya que de este trabajo depende el éxito de nuestra solución.

- a) **Tratar de iniciar la prueba** de un programa con una mentalidad sabotadora, casi disfrutando la tarea de encontrar algún error.
- b) **Sospechar de todos los resultados** que le arroje la solución, con lo cual deberá verificar todos.
- c) **Considerar todas las situaciones** posibles normales y aún algunas de las anormales.

### Verificación de algoritmos, ¿Que és?

Nos referimos a la comprobación del correcto funcionamiento del pseudocódigo planteado.

Es posible que al realizar la verificación del programa o partes del programa descubramos defectos que nos obliguen a volver a la parte de desarrollo. Las verificaciones, aunque tienen momentos principales, también es habitual que se extiendan a lo largo de las **fases de desarrollo, programación y mejora**.



## 3.2. Tipos de errores

### Errores de compilación

Se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser errores de sintaxis. Si existe un error de sintaxis, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.

### Errores de ejecución

Estos errores se producen por instrucciones que la computadora puede comprender pero no ejecutar. Ejemplos típicos son: división entre cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.

### Errores lógicos

Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo puede advertirse el error por la obtención de resultados incorrectos o no deseados. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.

## 4. Documentación

Es la guía o comunicación escrita en sus variadas formas, ya sean en enunciados, procedimientos, dibujos o diagramas. A menudo un programa escrito por una persona, es usado por muchas otras.

Por ello la documentación sirve para ayudar a comprender o usar un programa o para facilitar futuras modificaciones por parte de los programadores (mantenimiento). Debe presentarse en tres formas con respecto al programa, en forma externa, dentro del programa de manera interna y al usuario final.



La primera debe estar integrada por los siguientes elementos:

- a) Descripción del problema
- b) Nombre del autor
- c) Diagrama de flujo y/o pseudocódigo
- d) Lista de variables y constantes
- e) Codificación del programa

### Documentación externa

Incluye los aspectos técnicos del programa.

### Documentación interna

Constituyen los comentarios o mensajes que agregan al código, para hacer más claro el entendimiento del proceso. A la documentación para el usuario se le conoce como manual del usuario. En este manual no existe información de tipo técnico, sino la descripción del funcionamiento del programa.



En resumen, si en su momento dijimos que aprender a desarrollar algoritmos eficientes es aprender a programar, diremos ahora que aprender a verificar algoritmos es aprender a verificar programas.

## 5. Prueba de escritorio para validar algoritmos.



Es una herramienta útil para comprender cómo funciona una estructura, ya que nos permite ver y validar cómo funciona un algoritmo que diseñamos y buscar posibles errores.

Son [simulaciones del comportamiento](#) de un algoritmo que permiten determinar la validez del mismo.

Consisten en:

- generar una tabla con tantas columnas como variables tenga nuestro algoritmo y
- seguir las sentencias o instrucciones de nuestro algoritmo completando los valores correspondientes a medida que se van modificando.

Con esto podemos detectar:

- errores en tiempo de ejecución,
- errores de lógica,
- o bien para mejorar el algoritmo pensado.

Para poder llevar a cabo las [pruebas de escritorio](#), haremos previamente casos de prueba o lotes de prueba, estas son posibles situaciones de datos de entrada que tendrá que resolver nuestro programa y conocer con qué valor o resultado debe finalizar.

Por ejemplo, si tuviésemos que desarrollar un algoritmo en el cual se le pida al usuario ingresar 2 números enteros y obtener el resultado de dividir el primer número ingresado por el segundo,



Un posible lote de prueba sería:

numeroIngresado1: 20

numeroIngresado2: 5

Resultado esperado: 4



Otro posible lote de prueba sería:

numeroIngresado1: 10

numeroIngresado2: 4

Resultado esperado: 2.5



Siempre es recomendable considerar distintos escenarios como para testear nuestro algoritmo y ver cómo se comporta.

Una vez que tenemos los lotes de prueba, empezaremos a realizar la prueba de escritorio y para ello dijimos que vamos a colocar en una tabla las variables que tenga nuestro algoritmo.

# 5.1. Nuestro algoritmo


Nuestro algoritmo es el siguiente:

```
Algoritmo division
  Definir numeroIngresado1,numeroIngresado2 Como Entero
  Definir resultado Como Real
  Escribir "Ingresar el primer numero: "
  Leer numeroIngresado1
  Escribir "Ingresar el segundo numero: "
  Leer numeroIngresado2
  resultado = numeroIngresado1 / numeroIngresado2
  Escribir "El resultado de dividir: ",numeroIngresado1, " en ",numeroIngresado2, " es: ",resultado
FinAlgoritmo
```

La tabla para la prueba de escritorio quedaría:

numeroIngresado1	numeroIngresado2	resultado	Se muestra por pantalla

Y seguimos las instrucciones exactamente como nos indica nuestro algoritmo y vamos completando una nueva fila por cada sentencia que se va ejecutando.



Recordemos nuestro primer lote de prueba:

numeroIngresado1: 20

numeroIngresado2: 5

Resultado esperado: 4

- La primer sentencia del algoritmos es: escribir “Ingresar el primer numero” quedando nuestra tabla:

numeroIngresado1	numeroIngresado2	resultado	Se muestra por pantalla
			Ingresar el primer número:

- La siguiente sentencia es leer desde el teclado un número que ingrese el usuario y se almacenará en la variable numeroIngresado1 quedando nuestra tabla:

numeroIngresado1	numeroIngresado2	resultado	Se muestra por pantalla
			Ingresar el primer número:
20			

- Luego, el algoritmo solicita un segundo número y se ingresará en la variable [numeroIngresado2](#) quedando la tabla de la siguiente manera:

numeroIngresado1	numeroIngresado2	resultado	Se muestra por pantalla
			Ingresar el primer número:
20			
			Ingresar el segundo número:
	5		

- Luego, la variable resultado recibe el resultado de realizar la división de [numeroIngresado1](#) con [numeroIngresado2](#), con lo cual recibe el valor 4.

numeroIngresado1	numeroIngresado2	resultado	Se muestra por pantalla
			Ingresar el primer número:
20			
			Ingresar el segundo número:
	5		
		4	

- Y por último, se muestran por [pantalla los valores de las variables](#):

numeroIngresado1	numeroIngresado2	resultado	Se muestra por pantalla
			Ingresar el primer número:
20			
			Ingresar el segundo número:
	5		
		4	
			El resultado de dividir: 20 en 5, es: 4

Se recomienda hacer lotes de prueba con 1 o más casos extremos. Entendemos por casos extremos situaciones que casi nunca podrían suceder pero debemos asegurarnos que nuestro algoritmo las controle.



En nuestro algoritmo, un caso extremo sería el siguiente.

Lote de prueba - caso extremo:

numeroIngresado1: 20

numeroIngresado2: 0

Ese lote de prueba **generará un error en tiempo de ejecución** ya que no se puede dividir por cero.

```

PSeInt - Ejecutando proceso DIVISION
*** Ejecución Iniciada. ***
Ingresar el primer numero:
> 20
Ingresar el segundo numero:
> 0
Lin 8 (inst 1): ERROR 296: Division por cero
  
```

Lo que debemos hacer en caso de detectar errores,

- es modificar nuestro algoritmo para solucionar el inconveniente y luego,
- realizar una nueva prueba de escritorio.

El algoritmo contemplando división por cero queda:

```
Algoritmo division
Definir numeroIngresado1,numeroIngresado2 Como Entero
Definir resultado Como Real
Escribir "Ingresar el primer numero: "
Leer numeroIngresado1
Escribir "Ingresar el segundo numero: "
Leer numeroIngresado2
Si numeroIngresado2 ≠ 0 Entonces
    resultado = numeroIngresado1 / numeroIngresado2
    Escribir "El resultado de dividir: ",numeroIngresado1, " en ",numeroIngresado2, " es: ",resultado
SiNo
    Escribir "No se puede realizar una división por cero."
FinSi
FinAlgoritmo
```



Te proponemos como práctica modificar la tabla del lote de prueba generada anteriormente.