

Lógica de Programación. Nivel III

junio, 2019



Objetivos del nivel

- Entender la modularidad y su importancia
- Aprender a crear módulos reusables
- Comprender el paso de parámetros
- Entender el uso de contadores y acumuladores

Prerrequisitos del nivel

- Lógica de Programación Nivel II

Acerca de este manual

Este manual pertenece al Centro de Asesoramiento y Desarrollo Informático C.A. (CADI F1). Para obtener más información sobre este u otros cursos visite nuestro sitio Web www.cadif1.com, escribanos a la dirección de correo cadi@cadif1.com o visítenos en nuestra sede ubicada en la Av. Pedro León Torres con calle 59, Centro Comercial Sotavento, piso 2 oficina 27, Barquisimeto estado Lara, Venezuela. Tlf. 0251-7179247, 0251-4410268.

Las marcas mencionadas en este manual son propiedad de sus respectivos dueños.
Copyright 2019. Todos los derechos reservados.



Contenido del nivel

Capítulo 1. Modularidad. Parte 1

- 1.1.- Modularidad y Módulos.
- 1.2.- Definición de Módulos.
- 1.3.- Llamado de Módulos.
- 1.4.- Cuerpo Principal y Modularidad.

Capítulo 2. Comunicación Entre Módulos. Parte 1

- 2.1.- ¿Qué es un Parámetro?.
- 2.2.- Parámetros Formales.
- 2.3.- Parámetros Actuales.
- 2.4.- Relación Entre Los Parámetros.

Capítulo 3. Comunicación Entre Módulos. Parte 2

- 3.1.- Comportamiento de Los Parámetros.
- 3.2.- Parámetros Por Valor.
- 3.3.- Parámetros Por Referencia.
- 3.4.- Módulos Con Ambos Parámetros.

ACADEMIA DE SOFTWARE

Capítulo 4. Modularidad. Parte 2

- 4.1.- Módulos y Jerarquización.
- 4.2.- Jerarquización y Parámetros 1.
- 4.3.- Jerarquización y Parámetros 2.

Capítulo 5. Modularidad. Parte 3

- 5.1.- Creación de Sub-módulos.
- 5.2.- Reusabilidad Con Parámetros.
- 5.3.- Ventajas de la Modularidad.

Capítulo 6. Variables: Globales y Locales

- 6.1.- Variables Globales.
- 6.2.- Variables Locales.
- 6.3.- Ámbito de Las Variables.

Capítulo 7. Algoritmos Con Menús

- 7.1.- Menú y Opciones.
- 7.2.- Incorporación Del Según.
- 7.3.- Incorporando el Ciclo.

Capítulo 8. Módulos Como Funciones

- 8.1.- Funciones Externas y Modularidad.
- 8.2.- Declaración de Funciones Externas.
- 8.3.- Llamado de Funciones.

Capítulo 9. Mayores y Menores

- 9.1.- ¿Qué implica determinar el Mayor y Menor Valor?.
- 9.2.- Determinación de Mayores.
- 9.3.- Determinación de Menores.

Capítulo 10. Contadores y Acumuladores

- 10.1.- Generalidades Sobre Contadores y Acumuladores.
- 10.2.- Concepto de Contador.
- 10.3.- Contador General y Específico.
- 10.4.- Concepto de Acumulador.
- 10.5.- Acumulador General y Específico.

Capítulo 11. Promedios y Porcentajes 2

- 11.1.- Introducción a Cálculos Usando Ciclos.
- 11.2.- Promedio.
- 11.3.- Porcentajes.

Capítulo 12. Estructuras Repetitivas Dobles. Parte 1

- 12.1.- Estructura Repetitiva Anidada.
- 12.2.- Combinación de Ciclos Anidados.
- 12.3.- Lugar Para Entradas y Salidas.

Capítulo 13. Estructuras Repetitivas Dobles. Parte 2

- 13.1.- Contadores y Acumuladores en doble Ciclos.
- 13.2.- Promedios y Porcentajes en doble Ciclos.

Capítulo 14. Búsqueda en Arreglos

- 14.1.- Generalidades de la Búsqueda en Arreglos.
- 14.2.- Buscar un Elemento Único.
- 14.3.- Buscar un Elemento Repetido.

Capítulo 15. Procesando Valores en Arreglos

- 15.1.- Promedio Usando Arreglos.
- 15.2.- Determinando Mayores y Menores Usando Arreglos.
- 15.3.- Determinar Mayores.
- 15.4.- Determinar Menores.

Capítulo 1. MODULARIDAD. PARTE 1

1.1.- Modularidad y Módulos

La solución de un problema complejo puede requerir muchos pasos...

Por ello, a mayor complejidad en el problema, se hace más difícil crear el algoritmo que lo resuelva, razón por la cual, se perfila necesario aplicar alguna estrategia para simplificar la resolución del problema.

La estrategia o táctica más idónea, es dividir el problema inicial en sub-problemas que sean más fáciles de resolver. A este método se le denomina Divide y Vencerás. La aplicación computacional de esa táctica implica programar modularmente.

En general, la MODULARIDAD es la cualidad que tiene un sistema de ser estudiado, visto y entendido, como la unión de varias partes que interactúan entre sí trabajando cooperativamente para alcanzar un objetivo común, por lo tanto cada una de ellas es necesaria para lograr lo esperado.

En el ámbito de la programación, un MÓDULO (también llamado sub-proceso, sub-rutina, sub-programa, procedimiento o función) es cada una de las partes independientes en que se divide un programa.

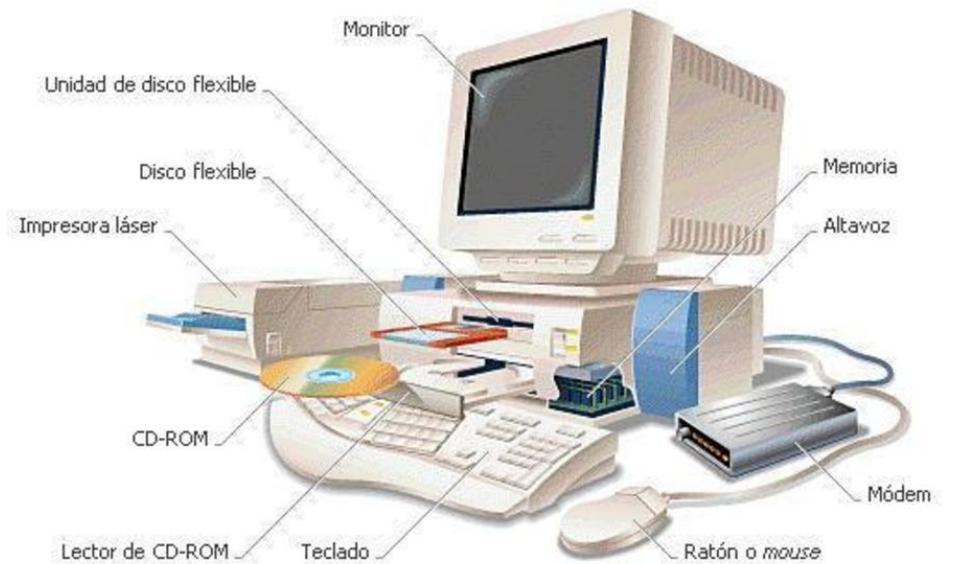
Cada módulo contiene un conjunto de instrucciones que están separadas del programa principal.

Un módulo debe poder funcionar como una caja negra, es decir, interactuar con el resto del programa cumpliendo con "hacer lo que debe", sin dar importancia a "cómo lo hace", es decir, ser independiente funcionalmente del resto de los módulos y comunicarse con ellos según sea necesario.

La computadora es un ejemplo de un equipo hecho de forma modular, debido a que cada una de las partes que la conforman interactúan para lograr un fin común (permitir al usuario utilizar el software que está corriendo sobre el hardware).

Cada una de las partes que conforman al computador, es independiente de la otra y se comunican entre sí para trabajar en conjunto.

Muchos de estos módulos poseen sub-módulos que lo componen, por ejemplo: la tarjeta madre tiene conectado el CPU, la memoria, las unidades ópticas, etc. Por su parte, el disco duro, internamente está dividido en módulos, esto mismo ocurre con otros componentes.



Un módulo debe resolver un problema específico (pequeño) del gran problema, por lo cual, idealmente un módulo debería tener un objetivo primario, lo cual se tratará en lo sucesivo como SINGULARIDAD DE PROPÓSITO.

Si dentro de un mismo módulo, se colocan muchas instrucciones relacionadas con el procesamiento de datos, el módulo tenderá a hacerse largo, en consecuencia será más difícil de examinar y mantener. Ésto evidenciaría que dicho módulo es multipropósito, y no se estaría sacando el máximo provecho de programación modular.

1.2.- Definición de Módulos

Para desarrollar un programa modular, los módulos deben ser definidos. La definición de un módulo, consiste en enmarcar bajo un nombre el conjunto de instrucciones que lo componen.

Los nombres dados a los módulos deben ser significativos.

No debería haber varios módulos con el mismo nombre dentro del mismo programa (esto tiene su excepción en la Programación Orientada a Objetos).

Cada lenguaje de programación establece una forma de definir los módulos.

En pseudocódigo de Pselnt, se usa un par de palabras reservadas para enmarcar las instrucciones de un módulo, lo cual implica que se está definiendo este tipo de bloque de código.

Los pasos para definir un módulo en Pselnt son los siguientes:

- 1) Colocar la palabra reservada de apertura (SubProceso o SubAlgoritmo),
- 2) Colocar el nombre del módulo, precedido por un espacio en blanco,
- 3) Sin dejar espacio, se colocan entre paréntesis, el(los) parámetro(s). Éste término se refiere a un dato que el módulo "necesita para algo" y se desarrolla en profundidad más adelante.
- 4) Se colocan las instrucciones que conforman el cuerpo del módulo, y permiten lograr su objetivo.
- 5) Finalmente, se coloca la palabra reservada de cierre (FinSubproceso o FinSubAlgoritmo).

A continuación, se ejemplifica la definición de 2 módulos en Pselnt (sin cuerpo) usando las combinaciones de las palabras claves referidas:

```
SubProceso modulo1()
    // aquí van las instrucciones que
    // se van a ejecutar dentro del módulo
FinSubProceso

SubAlgoritmo modulo2()
    // aquí van las instrucciones que
    // se van a ejecutar dentro del módulo
FinSubAlgoritmo
```

En pseudocódigo, los módulos son definidos normalmente antes del cuerpo principal.

El orden en que se definen es relevante en algunos lenguajes de programación, cuando los módulos se "usan" entre ellos.

Lo más recomendable para nombrar los módulos es que se utilicen verbos en infinitivo (finalizan en ar,er,ir), por ejemplo: calcular_montos, mostrar_salidas, etc.

A continuación, se aprecian una serie de módulos relacionados con un algoritmo Rutina Diaria para Salir:

Subproceso vestir()

buscarRopa
plancharRopa
ponerRopa

FinSubproceso**Subproceso comer()**

calentarComida
servirComida
consumirComida

FinSubproceso**Subproceso baniar()**

quitarRopa
mojar_enjabonar
secarConToalla

FinSubproceso

1.3.- Llamado de Módulos

Definir los módulos es parte esencial de la modularidad, así como también lo es el "Llamado o Invocación a los Módulos"; ello implica colocar una "instrucción particular" para indicar que se necesita que se ejecuten las instrucciones que están contenidas dentro del módulo.

Un módulo se llama o invoca colocando en una línea su nombre especificando los parámetros si los tuviere, esto se considera una instrucción. Cabe destacar que si un módulo no es llamado, no se ejecuta.

Los módulos pueden llamarse desde:

- El cuerpo principal del algoritmo. El orden en que se crean los módulos es irrelevante.

- Otros módulos del algoritmo. En algunos lenguajes de programación, el módulo que es llamado debe estar definido antes del módulo que lo llama.

Para que un módulo se ejecute, se le debe llamar desde donde sea conveniente y de hace colocando el nombre del módulo seguido de los parámetros (si los tiene).

A continuación, se verán varios aspectos de la modularidad exemplificados con el algoritmo modular de Rutina Diaria para salir.

En la imagen se exemplifica como llamar a los módulos (sin parámetros) desde el cuerpo principal del programa:



Subproceso vestir()

- buscarRopa
- plancharRopa
- ponerRopa

FinSubproceso**Subproceso comer()**

- calentarComida
- servirComida
- consumirComida

FinSubproceso**Subproceso baniar()**

- quitarRopa
- mojar_enjabonar
- secarConToalla

FinSubproceso**Proceso Rutina_Diaria_Salir**

// Cuerpo Principal

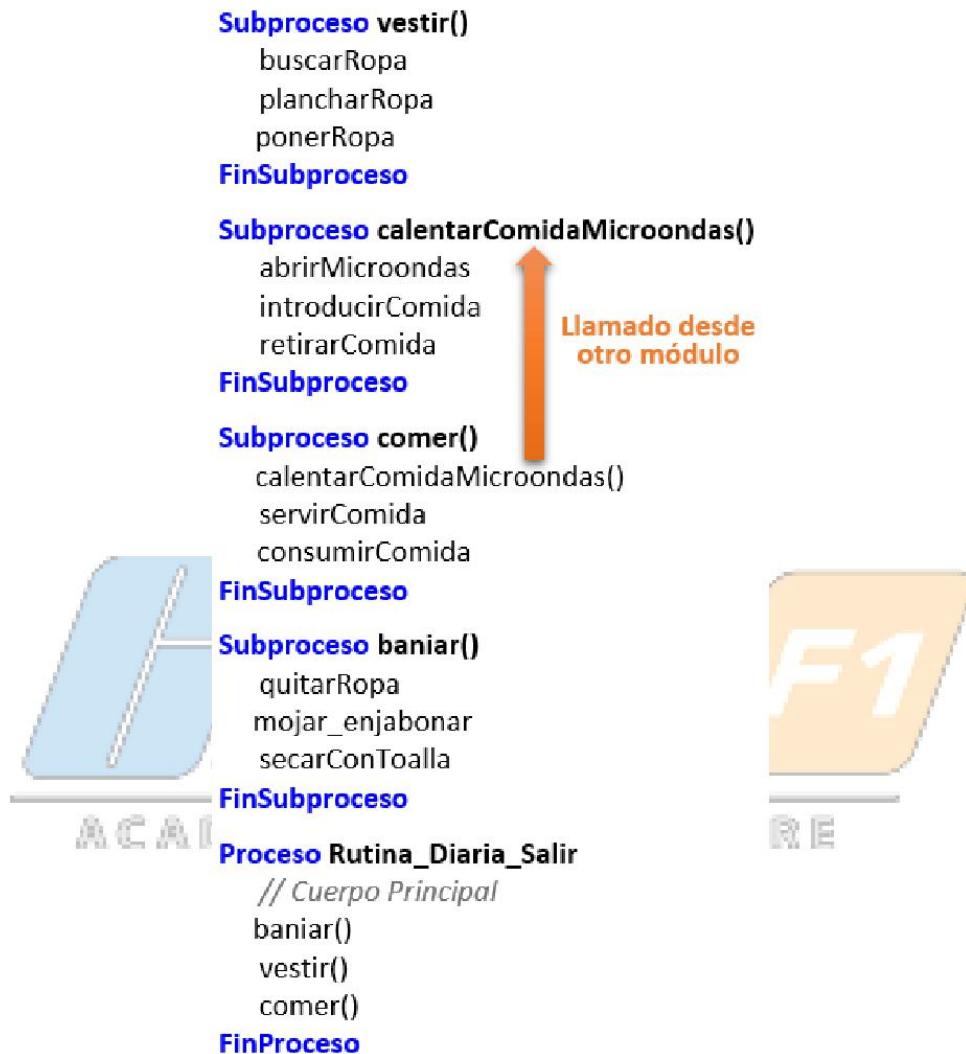
- baniar()
- vestir()
- comer()

**Llamados desde el
cuerpo principal**

FinProceso

Aquí se ejemplifica como llamar un módulo carente de parámetros desde otro módulo.

Nótese que la 1era acción del módulo "comer" es el llamado a un módulo "calentarComidaMicroondas" que está definido antes de él.



Un módulo puede ser llamado a ejecutarse varias veces. Esto puede hacerse desde el cuerpo principal o desde otros módulos (incluso en un mismo módulo puede llamarse varias veces a un módulo particular). A esto se le conoce como Reusabilidad (su implementación se verá en detalle más adelante).

Por lo anterior, no es necesario definir el módulo las veces que se necesite ejecutarlo. Es suficiente con definirlo una vez, y llamarlo tantas veces como sea necesario.

```

Subproceso vestir()
  buscarRopa
  plancharRopa
  ponerRopa
FinSubproceso

Subproceso calentarComidaMicroondas()
  abrirMicroondas
  introducirComida
  retirarComida
FinSubproceso

Subproceso comer()
  calentarComidaMicroondas()
  servirComida
  consumirComida
FinSubproceso

Subproceso cepillarDientes()
  untarPastaDentalCepillo
  cepillarDientes
  enjuagarBoca
FinSubproceso

Proceso Rutina_Diaria_Salir
  // Cuerpo Principal
  cepillarDientes()
  vestir()
  comer()
  cepillarDientes()
FinProceso

```

The diagram illustrates the concept of reusability. It shows two processes: 'RCA' (represented by a blue trapezoid icon) and 'F1' (represented by an orange trapezoid icon). A curved orange arrow labeled 'Reusabilidad' points from the 'cepillarDientes()' subprocedure within the 'Rutina_Diaria_Salir' process to its occurrence within the 'F1' process.

1.4.- Cuerpo Principal y Modularidad

El código fuente de un programa básico se escribe en una sección denominada CUERPO PRINCIPAL del programa. En algunos lenguajes de programación se conoce como el "main".

En la programación secuencial, todas las instrucciones de los programas se colocan en dicho bloque, lo que implica que cuando un algoritmo tiene muchas líneas de código podría dificultarse entenderlo, mejorarlo y hacerle correcciones.

Cuando se inicia la ejecución de un programa, se ejecutan primeramente las instrucciones que están en esta sección.

Ahora bien, cabe preguntarse... ¿Qué pasa con el control de ejecución cuando el programa es modular?, he aquí lo que sucede:

Al ejecutarse un programa, se ejecuta cada instrucción que esté en el cuerpo principal. En él puede haber instrucciones que no impliquen el llamado a un módulo.

Pero, cuando el cuerpo principal llama a un módulo, se pasa el control de ejecución del programa a ese módulo y se ejecuta cada instrucción contenida en el mismo.

Cuando se ejecuta la última instrucción del módulo, se devuelve el control de ejecución al cuerpo principal, para que ejecute la siguiente instrucción.

IMPORTANTE: Si el llamado a un módulo se origina en otro módulo, el control de ejecución se maneja entre ellos.

ACADEMIA DE SOFTWARE

Ahora bien, en la programación modular se busca que en el cuerpo principal se coloque la menor cantidad de instrucciones posibles. De manera que, la mayoría de las instrucciones estén en los módulos.

No existe una regla general sobre crear programas modulares, ya que se pueden crear primero los módulos y luego colocar las instrucciones que estarán en el cuerpo principal o viceversa.

En conclusión, un algoritmo o programa modular estaría constituido por uno o varios módulos y un único cuerpo principal.

Capítulo 2. COMUNICACIÓN ENTRE MÓDULOS. PARTE 1

2.1.- ¿Qué es un Parámetro?

Todo lo que existe en un módulo es inaccesible al resto de los módulos o al programa principal. Por esta razón, la programación modular exige una comunicación entre el módulo llamador y el módulo llamado. Esta comunicación se realiza a través de unas variables de enlace que se denominan parámetros.

Un PARÁMETRO o ARGUMENTO es un dato que se envía o que es retornado por un módulo del programa.

Un módulo puede:

- no tener parámetros,
- tener un solo parámetro o
- tener varios parámetros.

La cantidad de parámetros del módulo dependerá del objetivo que cumpla el módulo.

Para lograr la "Singularidad de Propósito" es altamente probable que se necesiten datos provenientes de otros módulos.



Por ejemplo, si se necesita crear un módulo para enviar mensajes de texto vía SMS, se definiría un módulo con el nombre "EnviarMensaje", el cual necesitará recibir al menos 2 parámetros:

- el número de teléfono al cual se va a enviar el mensaje, y
- el mensaje a enviar

Estos dos (2) datos pueden provenir de un módulo que le solicite éstos al usuario, o de un módulo que los lea de una fuente de datos externa (ej.: base de datos).

Además, el módulo "EnviarMensaje" podría retornar un parámetro que indique si el mensaje fue enviado exitosamente o no.

Esto reafirma que, los parámetros son variables cuyos valores son captados o determinados dentro de un módulo y son retornados para ser usados por otros módulos.

El uso de parámetros en la modularidad permite que la reusabilidad de código sea más provechosa, ya que se puede usar el mismo código en diversas situaciones que son idénticas, donde lo único que cambia son los datos involucrados.

En el caso del envío de mensajes, si se necesita enviar 100 mensajes, el procedimiento es el mismo, lo único que cambiará serán los números de teléfono destino y el posiblemente el mensaje a enviar.

Sin reusabilidad, para enviar todos los mensajes, se tendría que escribir el mismo código 100 veces, donde lo único que cambiaría serían los dos (2) datos necesarios: número de teléfono y mensaje. Analizando este ejemplo es evidente que, la modularidad implementando reusabilidad, permite minimizar la cantidad de líneas de código que conforman el programa, lo cual redunda en ahorro de tiempo.

2.2.- Parámetros Formales

Cuando se define un módulo, si se determina que éste necesita parámetros o retorna parámetros, éstos deben indicarse en la definición del módulo. A esto se le llama "Definición de los Parámetros". Los parámetros que se colocan en la definición de los módulos se denominan PARÁMETROS FORMALES.

Para definir dichos parámetros, a continuación del nombre del módulo, se debe colocar a continuación su(s) parámetro(s) encerrados entre paréntesis (). Si están en juego varios parámetros, se separan usando comas (,). En los lenguajes de programación, además se debe especificar el tipo de dato de cada parámetro.

La forma general de definir los parámetros en pseudocódigo (no PseInt) es la siguiente:

```
SubRutina nombreModulo (parametro1 : tipoDato, parametro2 : tipoDato)
    ....
FinSubRutina
```

En PseInt no es necesario colocar el tipo de dato del parámetro. Por lo tanto, la forma general de declarar los parámetros es la siguiente:

```

SubProceso ejemplo1 (parametro1, parametro2, parametro3, ..., parametroN)
.....
FinSubproceso

```

Para el ejemplo del envío de mensajes, la definición del módulo es la siguiente:

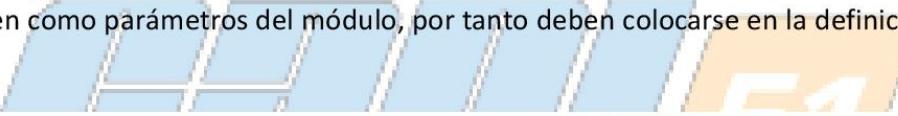
```

SubProceso EnviarMensaje (nroTelefono,mensaje)
.....
FinSubproceso

```

Recuérdese que también se puede utilizar la combinación de palabras reservadas SubAlgoritmo/FinSubAlgoritmo.

Como ejemplo se tiene un módulo encargado de solicitar 2 datos al usuario. Éstos se establecen como parámetros del módulo, por tanto deben colocarse en la definición del módulo:



```

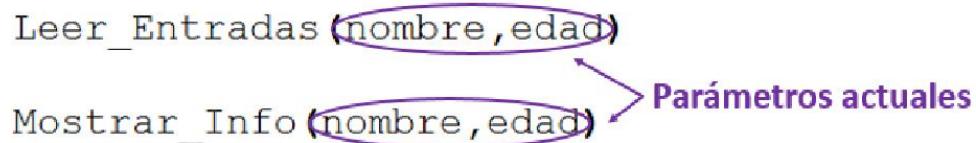
Subproceso Leer_Entradas (nombre, edad)
Mostrar "Ingresé nombre"
Leer nombre
Mostrar "Ingresé edad"
Leer edad
FinSubProceso

```

Parámetros formales

2.3.- Parámetros Actuales

Cuando un módulo tiene parámetros en su definición (Parámetros Formales), el llamado a ese módulo también debe tener parámetros. En el llamado se deben colocar los parámetros que el módulo necesita. A estos parámetros se les denomina PARÁMETROS ACTUALES. A diferencia de lo que pudiera requerirse con los parámetros formales, en los actuales no se debe colocar el tipo de dato del parámetro. Véase los siguientes ejemplos:



Los parámetros actuales pueden ser variables o valores literales dependiendo del tratamiento que se le dé a cada parámetro, lo cual se verá en el próximo capítulo. En el caso de que el valor literal sea alfanumérico, se debe hacer uso de las comillas dobles o comillas simples. Véanse los siguientes ejemplos:

<pre> // Llamado al módulo usando variables // en los parámetros actuales Leer_Entradas(nombre,edad) SubProceso Leer_Entradas(nombre,edad) Mostrar "Ingrese nombre y edad" Leer nombre,edad FinSubProceso </pre>	<pre> // Llamado al módulo usando literales // en los parámetros actuales Mostrar_Info("Victoria",25) SubProceso Mostrar_Info(nombre,edad) Mostrar nombre " tiene " edad " años" FinSubProceso </pre>
--	---

2.4.- Relación Entre Los Parámetros

Entre los parámetros actuales y formales existe una relación muy estrecha, fundamentado en el hecho de que entre ellos se deben cumplir las siguientes 3 condiciones:

- 1.- Deben tener la misma cantidad: en la definición e invocación de un módulo debe haber la misma cantidad de parámetros. Ej.: si en la definición se colocan 5 parámetros formales, en el llamado al módulo deben colocarse obligatoriamente 5 parámetros actuales. Ni más, ni menos. De lo contrario, ocurrirá un error. Véase el ejemplo de errores por cantidad de parámetros...

```

Subproceso Leer_Entradas(nombre)
    Mostrar "Ingrese el nombre"
    Leer nombre
    Mostrar "Ingrese la edad"
    Leer edad
FinSubProceso

SubProceso Procesar(edad,bono)
    Si edad<18
        entonces bono=500
        sino bono=250
    FinSi
FinSubProceso

Subproceso Mostrar_Salidas(bono)
    Mostrar "Bono: " bono
FinSubProceso

Algoritmo Det_Bono
    Definir nombre Como Caracter
    Definir edad,bono como Entero
    Leer_Entradas(nombre,edad)
    Procesar(edad)
    Mostrar_Salidas(bono)
FinAlgoritmo

```

¡Error!. Hay más parámetros actuales que formales

¡Error!. Hay menos parámetros actuales que formales

Correcto

2.- Deben ser del mismo tipo: debe haber correspondencia de tipo de dato en la relación uno a uno entre los parámetros actuales y formales. Por ej., si el módulo recibe 2 parámetros tipo real, no se pueden colocar en el llamado del módulo 2 parámetros alfanuméricos. En el ejemplo se aprecia un error por no coincidir el tipo de dato:

```

Subproceso Leer_Entradas(nombre,edad)
    Mostrar "Ingrese el nombre"
    Leer nombre
    Mostrar "Ingrese la edad"
    Leer edad
FinSubProceso

Algoritmo Det_Bono
    Definir nombre Como Caracter
    Definir edad como Entero
    Leer_Entradas(edad,nombre)
    // llamado a otros módulos
FinAlgoritmo

```

Incompatibilidad de tipos

¡Error!

- 3.- Deben estar en el mismo orden: los parámetros formales y actuales deben ser indicados en el mismo orden. De no respetar el orden, aunque no necesariamente ocurrirá un error de verificación de código, los resultados del módulo muy probablemente serán incorrectos. Ej.: si un módulo define 2 parámetros reales, uno indica la estatura y el otro el peso, al ser llamado el módulo, deben colocarse los parámetros actuales en el mismo orden, 1ero la estatura y 2do el peso. Véase este ejemplo donde no se cumple con esta regla:

```

SubProceso Procesar(edad,bono)
    Si edad<18
        entonces bono=500
    sino bono=250
    FinSi
FinSubProceso

Algoritmo Det_Bono
    Definir nombre Como Caracter
    Definir edad,bono como Entero
    // Leer_Entradas(edad,nombre)
    Procesar(bono,edad)
FinAlgoritmo

```

Los parámetros no están en el mismo orden

¡Error!

No es necesario que los parámetros actuales y formales tengan el mismo nombre (aunque inicialmente esto sea una práctica común). Es decir, los parámetros formales y actuales pueden tener nombres distintos, aun cuando se refieren al mismo dato.

Para que al momento de la corrida se "sepa" a qué se refiere cada parámetro, es suficiente con cumplir las tres (3) condiciones referidas.

Ej.: se puede definir un módulo con los parámetros formales nombre, apellido, edad. Y al invocar este módulo se puede hacer usando los parámetros actuales nomb, ape, ed.

```
Subproceso Leer_Entradas(nombre,apellido,edad)
    Mostrar "Ingrese el nombre"
    Leer nombre
    Mostrar "Ingrese el apellido"
    Leer apellido
    Mostrar "Ingrese la edad"
    Leer edad
FinSubProceso

Algoritmo Ejemplo
    Definir nomb,ape Como Caracter
    Definir ed como Entero
    Leer_Entradas(nomb,ape,ed)
FinAlgoritmo
```

Capítulo 3. COMUNICACIÓN ENTRE MÓDULOS. PARTE 2

3.1.- Comportamiento de Los Parámetros

Para que un programa modular sea funcional, se debe decidir cuál será el tratamiento que se le dará a cada parámetro. Esto determinará su comportamiento en el contexto de la comunicación entre módulos. Por ello, en cada módulo se debe decidir como tratar a cada parámetro, eso conduce a establecer si un parámetro será:

- **POR VALOR:** significa que el módulo recibe sólo una copia del valor que tiene la variable, o sea que no la puede modificar.
- **POR REFERENCIA:** significa que se pasa la posición de memoria donde está guardada la variable, así el módulo puede saber que valor contiene, y además puede modificarlo.

IMPORTANTE:

El tratamiento que se le vaya a dar al parámetro se establece en los parámetros formales (definición del módulo), nunca en los parámetros actuales (llamado de los módulos).

3.2.- Parámetros Por Valor

ACADEMIA DE SOFTWARE

Un parámetro POR VALOR, también referido como "Parámetro de Entrada", es aquel cuyo valor, no sufrirá ningún cambio dentro del módulo. Para que un parámetro sea tratado Por Valor, no es necesario añadir nada en la definición (alguna palabra reservada o símbolo). Sin embargo, en PseInt, si se desea se puede usar la combinación de palabras "por valor" al lado del parámetro. Al tener definido uno o varios parámetros formales en un módulo, por omisión, son tratados "por valor".

```

Subproceso Despedir(Nombre)
    Mostrar ""
    Mostrar "Hasta luego... " Nombre " !"
    Mostrar ""
FinSubproceso

Subproceso Despedir(Nombre por valor)
    Mostrar ""
    Mostrar "Hasta luego... " Nombre " !"
    Mostrar ""
FinSubproceso

```

Parámetro de Entrada

*Ambas formas funcionan para manejar el parámetro **Nombre** por valor*

En el ejemplo del módulo "EnviarMensaje", los parámetros "numero" y "mensaje" serían "Parámetros de Entrada", porque ellos nunca serán modificados dentro del módulo .

Los parámetros por valor son usados frecuentemente en los módulos que muestran resultados, esto se exemplifica en este módulo para mostrar la información de facturación de una tienda que vende camisas:

```

Subproceso MostrarFactura(nomCli, dirCli, cantCami, precUnit, dscto, APagar)
    Mostrar "Cliente: " nomCli
    Mostrar "Dirección: " dirCli
    Mostrar "Cantidad de camisas: " cantCami
    Mostrar "Precio Unitario: " precUnit " Bs."
    Mostrar "Descuento: " dscto " Bs."
    Mostrar "Total a pagar: " APagar " Bs."
FinSubProceso

```

Parámetros de Entrada
(Por Valor)

Si un módulo tiene parámetros por valor en su definición, cuando se haga el llamado a éste, los parámetros pueden ser valores literales o variables que ya contengan un valor (bien sea por asignación o por lectura).

```

Subproceso MostrarFactura(nomCli,cantCami,precUnit)
    Mostrar "Cliente: " nomCli
    Mostrar "Cantidad de camisas: " cantCami
    Mostrar "Precio Unitario: " precUnit " Bs."
    Mostrar "Total a pagar: " cantCami*precUnit " Bs."
FinSubProceso

Algoritmo MostradoFacts
    // Llamado con valores literales
    MostrarFactura("Mario Vera",5,2500)

    // Asignación de valores a variables
    nombC='Ana Pérez'
    cantCa=7
    Mostrar "-----"
    // Llamado con 2 variables y 1 valor literal
    MostrarFactura(nombC,cantCa,2500)
FinAlgoritmo

```

Cliente: Mario Vera
 Cantidad de camisas: 5
 Precio Unitario: 2500 Bs.
 Total a pagar: 12500 Bs.

 Cliente: Ana Pérez
 Cantidad de camisas: 7
 Precio Unitario: 2500 Bs.
 Total a pagar: 17500 Bs.

3.3.- Parámetros Por Referencia

Un parámetro POR REFERENCIA, también referido como "Parámetro de Salida", es aquel cuyo valor es modificado dentro del módulo, y su nuevo valor debe "salir" para estar disponible a otro(s) módulo(s). La forma de indicar que un parámetro es de salida, varía de un lenguaje a otro. En PseInt, para que un parámetro sea de salida se deben colocar las palabras reservadas "Por Referencia" luego del identificador del parámetro.

ACADEMIA DE SOFTWARE

```

Subproceso Dar_Bienvenida(nombre Por Referencia)
    Mostrar "Ingrese su nombre" Parámetro de Salida
    Leer nombre
    Mostrar "Bienvenid@!" nombre
    Mostrar ""
FinSubProceso

```

Uno de los usos más clásicos de estos parámetros es el módulo donde se captan las entradas, debido a que es allí donde los parámetros tomarán el valor leído por el teclado, para que finalizada la ejecución del módulo, los valores guardados en esos parámetros estén disponibles a otros módulos.

```

Subproceso Leer_E entradas(nombre Por Referencia,edad Por Referencia)
    Mostrar "Ingrese nombre y edad"
    Leer nombre,edad
    Parámetros de Salida
        (Por Referencia)
    FinSubProceso

```

En el llamado a los módulos, los Parámetros por Referencia SOLO deben ser variables, nunca literales.

```

Subproceso Leer_E entradas(nombre Por Referencia,edad Por Referencia)
    Mostrar "Ingrese nombre y edad"
    Leer nombre,edad
    FinSubProceso

Algoritmo LeeMost_Datos
    Definir nombP1,nombP2 Como Caracter
    Definir edadP1 como Entero
    Leer_E entradas(nombP1,edadP1) ← Correcto
    // Procesamiento de datos de Persona 1
    Leer_E entradas(nombP2,27) ← |Error!...El 2do parámetro NO
    // Procesamiento de datos de Persona 2 es una variable, es un literal
    FinAlgoritmo

```



3.4.- Módulos Con Ambos Parámetros

ACADEMIA DE SOFTWARE

Un módulo puede tener sólo parámetros "Por Referencia", o sólo parámetros "Por Valor" o ambos.

```
Subproceso Leer_Entradas(nombre Por Referencia,edad Por Referencia)
    Mostrar "Ingrese el nombre" Sin Saltar
    Leer nombre
    Mostrar "Ingrese la edad" Sin Saltar
    Leer edad
FinSubProceso
```

Sólo parámetros por referencia

```
SubProceso Procesar(edad,bono Por Referencia)
    Si edad<18
        entonces bono=500
        sino bono=250
    FinSi
FinSubProceso
```

**Con ambos parámetros:
Por valor y por referencia**

```
Subproceso Mostrar_Salidas(nombre,bono)
    Mostrar nombre " debe percibir " bono " Bs."
FinSubProceso
```

Sólo parámetros por valor

```
Algoritmo Det_Bono
    Definir nombre Como Caracter
    Definir edad,bono como Entero
    Leer_Entradas(edad,nombre)
    Procesar(bono,edad)
    Mostrar_Salidas(nombre,bono)
FinAlgoritmo
```

Los módulos de proceso, en la mayoría de los casos, deben tener Parámetros de Entrada (por valor) y Salida (por referencia), ya que deben recibir datos o información, procesarla y "enviar" información resultante a otro(s) módulo(s).

No es necesario que los parámetros estén en un orden especial, es decir, no tienen que estar primero los parámetros "Por Referencia" y luego los que son "Por Valor" o viceversa. Incluso pudieran estar intercalados.

Sin embargo, es una práctica común, indicar primero los parámetros Por Valor y luego los parámetros Por Referencia.

Capítulo 4. MODULARIDAD. PARTE 2

4.1.- Módulos y Jerarquización

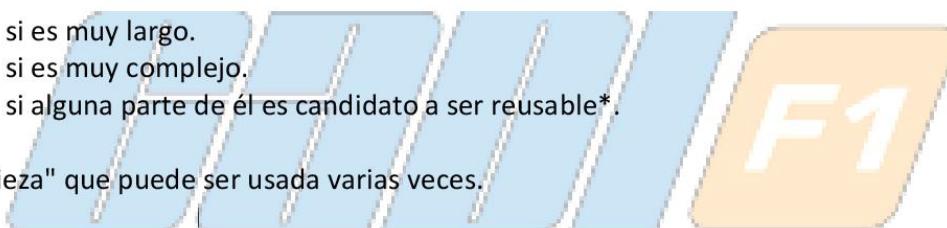
La cantidad de módulos en un programa dependerá de la complejidad y amplitud del problema.

Algunos módulos pueden dividirse en varios sub-módulos y así sucesivamente hasta que queden módulos indivisibles, en procura de que los módulos deben sean cortos, concretos y de poca complejidad.

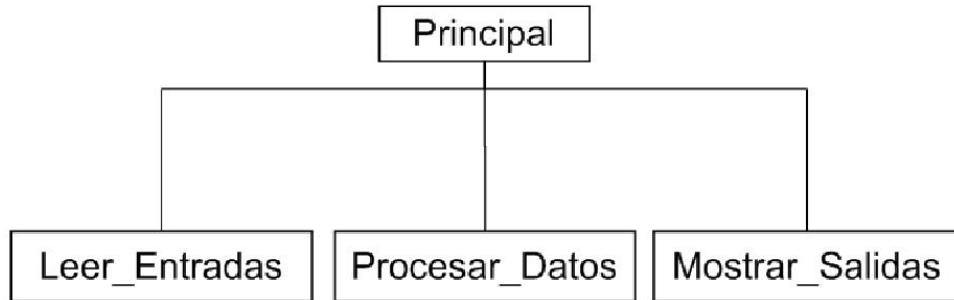
Los criterios para dividir un módulo en sub-módulos son subjetivos, pero se puede partir de los siguientes:

- si es muy largo.
- si es muy complejo.
- si alguna parte de él es candidato a ser reusable*.

(*) "pieza" que puede ser usada varias veces.

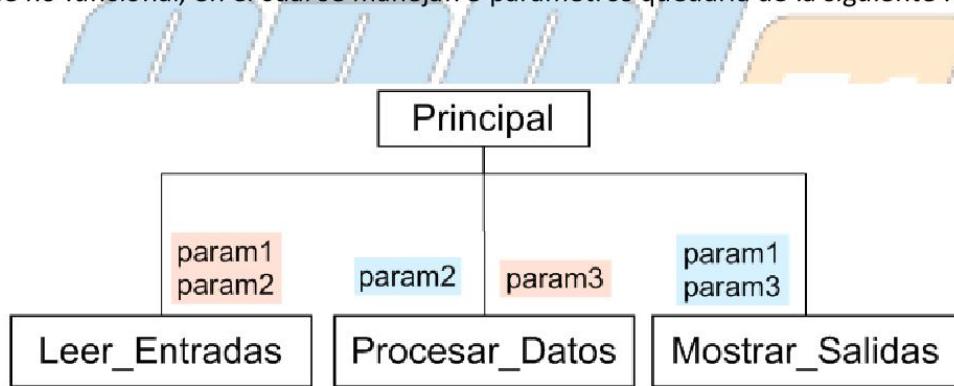


Para facilitar el análisis de un problema que se resolverá aplicando modularidad, la división puede plasmarse en un "Diagrama Jerárquico" que muestre la relación que existirá entre los módulos. En el caso de un problema sencillo deberían crearse al menos 3 módulos, nombrados según sus objetivos (entradas, proceso y salidas). En algunos casos, el módulo procesar_datos es el que tiende a ser sub-dividido debido a que es en él donde típicamente está la parte más compleja del programa. A continuación, se presenta un ejemplo de la división modular básica de un programa:



4.2.- Jerarquización y Parámetros 1

La mayoría de los módulos, debe contar con una cantidad de datos comunicados a través de los parámetros. A modo de ejemplo, el Diagrama Jerárquico de un programa modular simple no funcional, en el cual se manejan 3 parámetros quedaría de la siguiente forma:



Luego de crear el diagrama, se debe escribir el código fuente del programa (en el presente caso el algoritmo), haciendo que sea coherente con el mismo.

Así, el algoritmo modular básico (sin especificar el tratamiento de los parámetros) quedaría de la siguiente forma:

```
Subproceso Leer_Entradas(param1,param2)
    // Instrucciones para obtener entradas
FinSubProceso

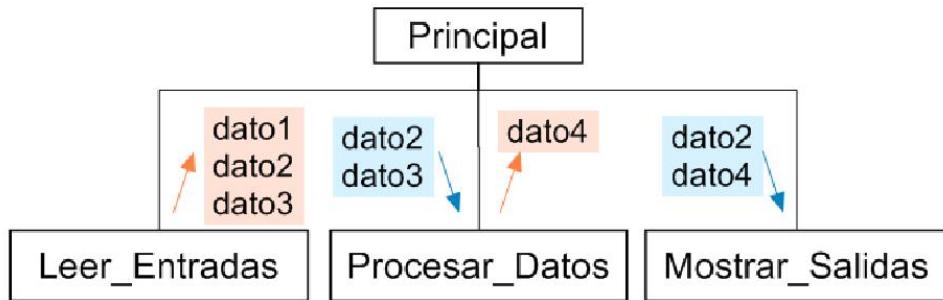
Subproceso Procesar_Datos(param2,param3)
    // Instrucciones para procesar datos
FinSubProceso

Subproceso Mostrar_Salidas(param1,param3)
    // Instrucciones para generar salidas
FinSubProceso

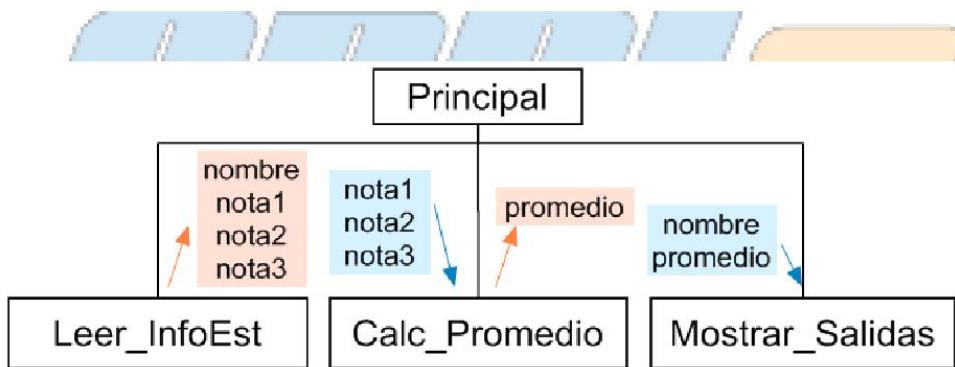
Algoritmo Modular_Generico
    // Definición de parámetros
    Leer_Entradas(param1,param2)
    Procesar_Datos(param2,param3)
    Mostrar_Salidas(param1,param3)
FinAlgoritmo
```

4.3.- Jerarquización y Parámetros 2

Para que el programa modular sea funcional, entre los módulos debe existir una comunicación que se hace a través de los parámetros. Esto tiene que ver con establecer distinción entre parámetros "por valor" o "por referencia" y esto se debe mostrar en el diagrama jerárquico, haciendo uso de flechas para distinguir lo que se obtiene de un módulo (por referencia/de salida) y lo que un módulo necesita (por valor/de entrada). Obsérvese este ejemplo:



Por ejemplo, con base en el siguiente enunciado: "Leer el nombre de un alumno y sus 3 notas parciales, calcular su nota promedio y mostrar el nombre y la nota promedio", el análisis concluye en el empleo de 3 módulos: Leer_InfoEst, Calc_Promedio y Mostrar_Salidas. El Diagrama Jerárquico en este caso sería el siguiente:



En el módulo Leer_InfoEst se solicitan los datos de entrada al usuario: nombre, nota1, nota2, y nota3. Estos deben salir de este módulo para ser usados en el 2do módulo.

En el módulo Calc_Promedio se reciben los datos nota1, nota2 y nota3, que vienen del módulo Leer_InfoEst. Con estos datos se calculará el promedio que debe salir de este módulo para ser usado en el módulo Mostrar_Salidas, donde dicha información será desplegado por pantalla junto con el nombre que viene del módulo Leer_InfoEst.

Un algoritmo funcional correspondiente al diagrama jerárquico del ejemplo anterior es el que se muestra a continuación. Nótese que los parámetros formales y actuales son diferentes.

```
Subproceso Leer_InfoEst(nombre Por Referencia,nota1 Por Referencia,  
                        nota2 Por Referencia, nota3 Por Referencia)  
    Mostrar "Ingrese el nombre del estudiante"  
    Leer nombre  
    Mostrar "Ingrese las 3 notas parciales sobre 20"  
    Leer nota1,nota2,nota3  
FinSubProceso

Subproceso Calc_Promedio(nota1,nota2,nota3,promedio Por Referencia)  
    promedio=(nota1+nota2+nota3)/3  
FinSubProceso

Subproceso Mostrar_Salidas(nombre,promedio)  
    Mostrar "El promedio de " nombre " fue: " promedio " ptos."  
FinSubProceso

Algoritmo Procesar_Estudiante  
    Definir nom como caracter  
    Definir nt1,nt2,nt3,prom_notas como real  
    Leer_InfoEst(nom,nt1,nt2,nt3)  
    Calc_Promedio(nt1,nt2,nt3,prom_notas)  
    Mostrar_Salidas(nom,prom_notas)  
FinAlgoritmo
```



ACADEMIA DE SOFTWARE

Capítulo 5. MODULARIDAD. PARTE 3

5.1.- Creación de Sub-módulos

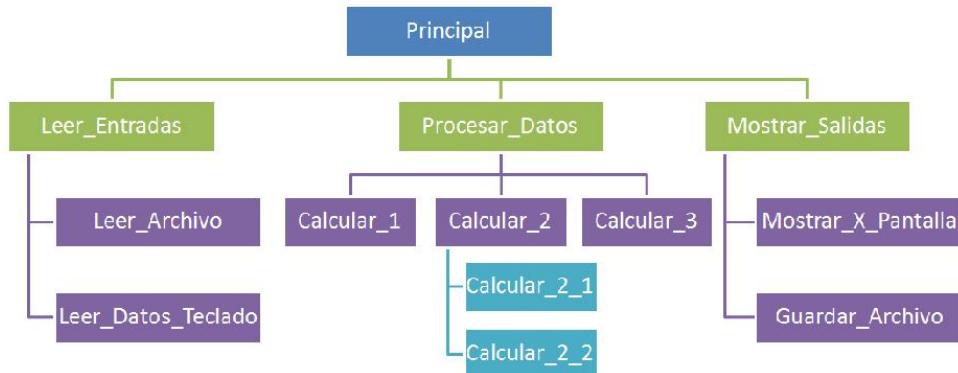
Cuando los problemas son más grandes o más complejos, será necesaria la sub-division del mismo en varios módulos. Dicha división en sub-módulos se hace considerando la complejidad y los requerimientos a satisfacer.

Esta división puede hacerse altamente granular, es decir, crear sub-módulos tantas veces como sea necesario hasta lograr que cada uno de ellos sea lo más sencillo posible, de manera que el módulo resulte indivisible funcionalmente.

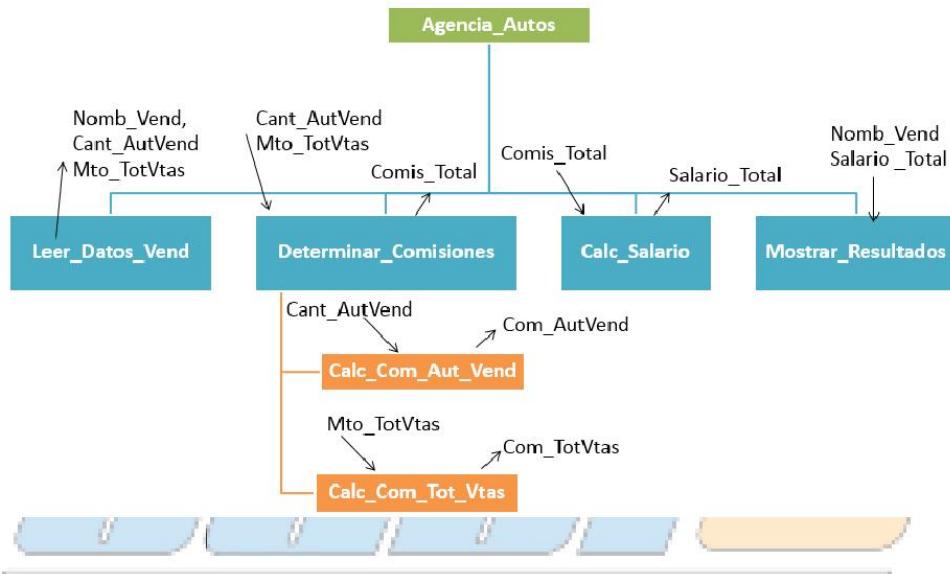
Cuando la subdivisión se hace granular, no siempre la invocación a los todos los módulos se hace desde el cuerpo principal del algoritmo.

En el siguiente Diagrama Jerárquico (sin parámetros) se muestra un ejemplo de un algoritmo genérico, cuya división se ha hecho más granular dada su complejidad.

Cabe destacar que una vez se especifican los parámetros, no todos se utilizan en el cuerpo principal, ya que algunos de ellos pueden tener un ámbito de uso que se limita a los módulos padre de los cuales derivan sus sub-modulos, nótese como esto aplica al módulo Calcular_2.



Véase el Diagrama Jerárquico del caso de la Agencia de Autos creando sub-modulos del módulo Determinar_Comisiones, dentro del cual gracias la división del trabajo en 2 sub-modulos se hacen más detalladas los cálculos de las comisiones parciales que determinan la comisión total del vendedor.



5.2.- Reusabilidad Con Parámetros DE SOFTWARE

La REUSABILIDAD consiste en crear piezas de código que se puedan usar varias veces a través del programa.

El empleo de parámetros es muy importante cuando se desean crear módulos reusables.

Ya se comprobó que no es estrictamente necesario que los parámetros formales y actuales de un módulo tenga el mismo nombre o identificador. Esto juega un papel importante en la reusabilidad, ya que se puede definir un módulo con parámetros, que pueda usarse varias veces pasándole valores distintos en los parámetros (actuales) en cada llamado. Solo hay que cumplir las 3 condiciones vistas con anterioridad.

En el ejemplo del módulo "EnviarMensaje" para enviar 100 mensajes a 100 números de teléfono diferentes, el módulo se definiría una vez y se reutilizaría 100 veces, por ej: se invocaría 100 veces con datos diferentes.

En este ejemplo se reusa el módulo Mostrar_Promedio para determinar y mostrar la información pertinente de 2 estudiantes:

```
Subproceso Mostrar_Promedio(nombre,nota1,nota2,nota3)
    Definir promedio como Real
    promedio=(nota1+nota2+nota3)/3
    Mostrar "Alumno: " nombre
    Mostrar "Promedio: " promedio " pts."
FinSubProceso
```

Estos llamados pueden hacerse desde el cuerpo principal o desde otro módulo

```
// llamado al módulo usando valores literales
→ Mostrar_Promedio("Juan",18,15,20)

Mostrar "Ingrese nombre " Sin Saltar
Leer nomb
Mostrar "Ingrese las 3 notas a promediar"
Leer nt1,nt2,nt3
// llamado al módulo usando nombres de variables
→ Mostrar_Promedio(nomb,nt1,nt2,nt3)
```

En este otro ejemplo se reusa el módulo Determ_Sueldo para calcular el sueldo que debe percibir un par de empleados:

```

Subproceso Determ_Sueldo(bono,ded,sueldoFinal Por Referencia)
    sueldoFinal = 80000 + bono - ded
FinSubProceso

Algoritmo Procesar_Empleados
    Definir bono,deduc,sdoF_E1,sdoF_E2 como real
    // Procesamiento para Empleado 1
    bono = 10500
    deduc = 2550
    Determ_Sueldo(bono,ded,sdoF_E1)
    Mostrar "Para el primer empleado " Sin Saltar
    Mostrar "el sueldo determinado es " sdoF_E1 " Bs."
    // Procesamiento para Empleado 2
    Determ_Sueldo(8000,1900,sdoF_E2)
    Mostrar "Para el segundo empleado " Sin Saltar
    Mostrar "el sueldo determinado es " sdoF_E2 " Bs."
FinAlgoritmo

```

5.3.- Ventajas de la Modularidad

En informática, la modularidad tiene muchas ventajas, entre ellas las más notables son las siguientes:

- Simplifica el diseño del algoritmo y el consecuente programa.
- Disminuye la complejidad de los algoritmos.
- Ahorra en tiempo de programación gracias a la reusabilidad del código.
- Facilita el mantenimiento del código.
- Favorece el trabajo en equipo.
- Facilita la depuración (identificación y resolución de errores).
- Facilita la prueba funcional del código.

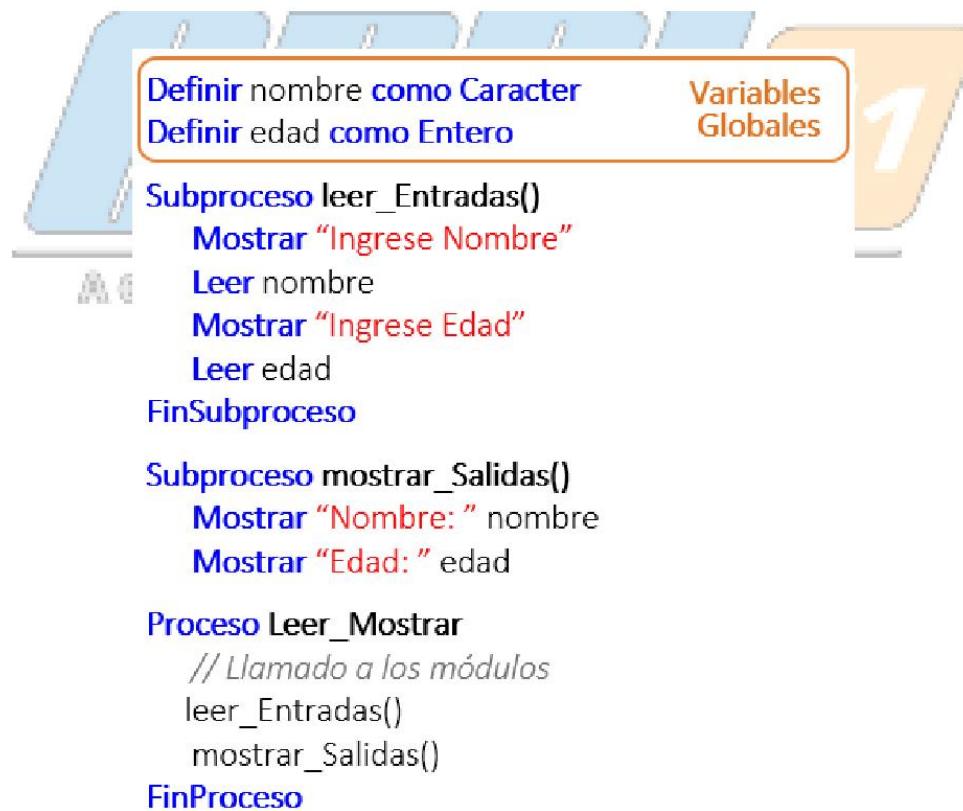
Capítulo 6. VARIABLES: GLOBALES Y LOCALES

6.1.- Variables Globales

Son variables que se declaran afuera de los módulos y pueden ser usadas en cualquier módulo del programa y/o en el cuerpo principal del programa. Una variable global ocupa espacio en la memoria RAM durante la ejecución de todo el programa.

El uso de variables globales no es recomendado, debido a que puede hacer que los programas sean más difíciles de entender y se sucedan ambigüedades.

Este sería un ejemplo de definición de variables globales en pseudocódigo, sin embargo, PSeInt no admite el uso de este tipo de variables:



6.2.- Variables Locales

Son variables que se declaran dentro de un módulo y solo pueden usarse en el módulo donde fueron declaradas. El objetivo de una variable local es apartar la memoria necesaria para las variables sólo en el momento en que se van a usar. El siguiente es un ejemplo de declaración de variables locales:

```
Subproceso calcularSueldo()  
    definir bono, aumento como real // variables locales  
  
        bono = nroHijos*5  
        aumento = sueldoBase * 0.1  
        sueldoNuevo = sueldoBase + aumento + bono
```

FinSubproceso

Nótese que `nroHijos` y `sueldoBase` no son variables locales, por lo tanto, se infiere que son variables globales.

Las variables "bono" y "aumento" son variables locales del módulo "calcularSueldo" y solo pueden ser usadas dentro de ese módulo.

Las variables locales son usadas normalmente, para hacer operaciones auxiliares previas a los cálculos o determinaciones finales del módulo dentro del cual están definidas.

IMPORTANTE:

Para entenderlo de un modo práctico, si se va a usar una variable, pero ella no está definida como parámetro, debe ser declarada como variable local del módulo que la utiliza.

También debe tenerse en cuenta que, dos o más módulos pueden tener variables locales con el mismo nombre sin que esto genere conflictos.

Cuando una variable X es local en el módulo "mod1", significa que dicho módulo es el "propietario" de dicha variable, y puede acceder a ella y modificarla, mientras se estén ejecutando instrucciones de ese módulo. Si cualquier otro módulo del programa necesita conocer el valor de la variable X, es "mod1" el que debe transferir el valor de X a través del paso de un parámetro por referencia.

6.3.- Ámbito de Las Variables

Una de las características de la programación modular es la independencia de los módulos. Esta independencia es importante, no sólo porque puede ser diseñado sin conocimiento del diseño de otros módulos, sino también porque la ejecución de un subproceso particular no afectará a los valores de las variables que pertenecen a otros subprocesos. Esto se vuelve más complejo en los lenguajes de programación que permiten el anidamiento de módulos (lo cual está fuera del alcance de este curso).

Por esta razón, es preciso definir un nuevo concepto de gran importancia: Ámbito de una variable.

Se conoce como "Ámbito de una variable" a la parte de un programa donde ésta es relevante, en otras palabras, todos aquellos lugares de un programa donde se puede utilizar la variable.

En relación al ámbito, se cumplen las siguientes reglas:

- Una variable local solo puede ser usada en el bloque (módulo) donde se declara.
- En un mismo bloque no se pueden duplicar los nombres de las variables.
- Una variable se puede declarar con el mismo nombre en bloques diferentes (módulos).
- Si una variable local tiene el mismo nombre de una variable global, al hacer referencia al nombre de la variable dentro del módulo, se accede a la variable local.

En el siguiente ejemplo se puede demostrar como se aplica el ámbito de las variables. Las salidas del siguiente programa serían:

50 , 20 , 25 (Cuerpo Principal)
 25 , 10 , 40 , 50 (Modulo A)
 30 , 45 , 60 , 10 (Modulo B)
 25 , 45 , 25 (Cuerpo Principal)

Definir x,y,z como entero // Variables Globales

Subproceso Modulo_A

Definir y,z,w como entero // Variables locales
 y = 10
 x = 25
 w = 50
 z = 40
 Mostrar x „, y „, z „, w

x al no ser una variable
 local, modifica la
 variable global con el
 valor 25 (x=25)

FinSubproceso

Subproceso Modulo_B

Definir x,t,z como entero // Variables locales
 x = 30
 y = 45
 z = 60
 t = 10
 Mostrar x „, y „, z „, t

y al no ser una variable
 local, modifica la
 variable global con el
 valor 45 (y=45)

FinSubproceso

Proceso

x = 50
 y = 20
 z = 25
 Mostrar x „, y „, z
 Modulo_A
 Modulo_B
 Mostrar x „, y „, z

FinProceso

Capítulo 7. ALGORITMOS CON MENÚS

7.1.- Menú y Opciones

En informática, un menú es una serie de opciones entre las cuales el usuario puede elegir para realizar determinadas tareas. Cada una de las cuales típicamente están asociadas a la invocación a módulos.

En su forma más simple, un menú se presenta en forma de lista, la cual se despliega mediante instrucciones de salida donde se muestra información sobre las opciones, junto con un mensaje indicándole al usuario que debe ingresar una opción:

```

*** Ejecución Iniciada. ***
MENÚ DE OPCIONES
1 - Ingresar Datos
2 - Realizar Cálculos
3 - Mostrar resultados
4 - Salir
Ingrese una opción...

```

Después de presentar el menú, y de requerirle al usuario que ingrese la opción de su preferencia, ésta debe captarse a través de una variable que se usará para determinar lo que ocurrirá a continuación:

```

*** Ejecución Iniciada. ***
MENÚ DE OPCIONES
1 - Ingresar Datos
2 - Realizar Cálculos
3 - Mostrar resultados
4 - Salir
Ingrese una opción...
>

```

```

*** Ejecución Iniciada. ***
MENÚ DE OPCIONES
1 - Ingresar Datos
2 - Realizar Cálculos
3 - Mostrar resultados
4 - Salir
Ingrese una opción...
Leer opc_menu

```

7.2.- Incorporación Del Según

Una vez leída la opción de menú, se debe dirigir el control de flujo del algoritmo usando una estructura de alternativa de selectiva múltiple (SEGÚN).

Por cada opción que implica ejecutar acciones, se le debe pasar el control de ejecución al módulo correspondiente. Nótese que este ejemplo no se han especificado los parámetros actuales, sin embargo, es clave tener presente que tanto ellos como los parámetros formales serán necesarios para posibilitar la comunicación entre los módulos.

Nótese que en el caso que sea ingresado una opción no válida, se le informa al usuario mediante un mensaje...a través de la opción "De Otro Modo" del Según.

```
Segun opc_menu hacer
    1: Leer_Datos
    2: Realizar_Cálculos
    3: Mostrar_Salidas
    4: mostrar "Ud. escogió salir del programa"
    De Otro Modo:
        mostrar "Opción incorrecta"
FinSegun
```

7.3.- Incorporando el Ciclo

Luego de ejecutada la acción asociada, se debe presentar nuevamente el menú para que el usuario pueda seleccionar una nueva opción. El ciclo que típicamente se usa a estos efectos es el "Repetir-Hasta", el cual estará controlado por el valor que tome la variable evaluada en el Según. El bucle se rompe cuando el usuario selecciona la opción para salir.

La estructura básica completa quedará de la siguiente manera:

```
Algoritmo Modulos_Y_Menu
    definir opc_menu Como Entero

    repetir
        mostrar "MENÚ DE OPCIONES"
        mostrar "1 - Ingresar Datos"
        mostrar "2 - Realizar Cálculos"
        mostrar "3 - Mostrar resultados"
        mostrar "4 - Salir"
        mostrar "Ingrese una opción..."
        Leer opc_menu
        Segun opc_menu hacer
            1: Leer_Datos
            2: Realizar_Cálculos
            3: Mostrar_Salidas
            4: mostrar "Ud. escogió salir del programa"
            De Otro Modo:
                |   mostrar "Opción incorrecta"
        FinSegun
    Hasta Que opc_menu = 4
FinAlgoritmo
```



ACADEMIA DE SOFTWARE

Capítulo 8. MÓDULOS COMO FUNCIONES

8.1.- Funciones Externas y Modularidad

En general, una función es una pequeña parte de un programa o software que realiza una tarea particular, devolviendo un resultado o valor.

Con anterioridad se vió que en Pselnt hay funciones internas básicas, que regularmente se encuentran implementadas en los lenguajes de programación, con el objeto de realizar operaciones típicas en los algoritmos o programas. Este es el caso de las funciones abs, azar, longitud, mayúsculas, minúsculas, subcadena, entre otras.

Cuando se vió este concepto en el Nivel 1, se citó que existían Funciones Externas que eran creadas por el programador.

Habiendo aprendido a implementar Modularidad, se puede crear cualquier función a discreción, la cual no será más que un módulo que se caracteriza por retornar un solo valor.

8.2.- Declaración de Funciones Externas

La declaración de una función externa es diferente a la de un módulo tradicional. Cabe destacar que cada lenguaje de programación tiene sus particularidades en cuanto a la declaración.

En el caso de Pselnt, la función se declara colocando la palabra Funcion (tambien se puede usar las palabras subprocesso o subalgoritmo), luego se debe colocar el nombre de la variable de retorno que se calculará o determinará, luego el operador de asignación y luego el nombre de la función, con los parámetros si los requiriese. Cabe destacar que las funciones deberían tener únicamente Parámetros por Valor. La función se "cierra" con la palabra reservada FinFuncion.

```

Funcion variable_de_retorno1 <- Nombre_Funcion1(param1)
    variable_de_retorno =param1
FinFuncion

Funcion variable_de_retorno2 = Nombre_Funcion2(param1,param2)
    variable_de_retorno =param1+param2
FinFuncion

```

En el siguiente ejemplo se puede observar la definición de una función, cuyo objetivo es calcular y devolver el doble de un número pasado a la función como un parámetro.

```

Funcion dobleDelNum <- CalcularDoble(num)
    dobleDelNum = num * 2
FinFuncion

```

En el siguiente ejemplo se puede ver la diferencia entre un módulo tradicional y un módulo definido como función, en cuanto a conformación y definición de parámetros.

En este ejemplo se puede notar cómo en el módulo tradicional, el parámetro "prom_notas" debe indicarse por referencia, en cambio, en la función, ese parámetro no existe debido a que ese es el valor es la variable de retorno que calculará la función:

```

Subproceso Calcular_NotaProm2(exam1,exam2,exam3,prom_notas Por Referencia)
    prom_notas=(exam1+exam2+exam3)/3
FinSubProceso

Funcion prom_notas <- Calcular_NotaProm1(exam1,exam2,exam3)
    prom_notas=(exam1+exam2+exam3)/3
FinFuncion

```

8.3.- Llamado de Funciones

El llamado de las funciones es particularmente diferente al de los módulos tradicionales, debido a que ellas retornan un valor. Éste puede usarse para mostrarse en pantalla,

almacenarse en una variable, para ser comparado, entre otras opciones, por ej: pasarlo por parámetro, incluirlo dentro de una operación aritmética, etc.

Algoritmo Uso_de_Funciones_Externas

```
// Mostrar en pantalla el valor que retorna la función
Mostrar nombre_funcion(parametro)

// Asignar a una variable el resultado de llamar a la función
variable = nombre_funcion(parametro)

// Llamado desde una sentencia selectiva o condicional
Si nombre_funcion(parametro)>15
|
FinSi

// Llamado desde un ciclo Repetir-Hasta
Repetir
|
Hasta Que nombre_funcion(parametro)>5

FinAlgoritmo
```

En el siguiente ejemplo se puede observar varios usos de una función llamada CalcularDoble llamada desde el cuerpo principal (main), para mostrar el resultado directamente, asignar el resultado de su llamado a otra variable, y evaluar lo que resulta de la función (dentro de una sentencia condicional).

```
Algoritmo LlamadoAFunCalcDoble
  Definir nro,doble Como Entero

  Mostrar "Ingrese un número" Sin Saltar
  Ler nro
  // Llamado desde una sentencia Mostrar
  Mostrar "El doble del número es: " nro " es " CalcularDoble(nro)
  // Asignación a una variable de lo que retorna una función
  doble = CalcularDoble(nro)
  // Llamado a la función desde una selectiva
  Si CalcularDoble(nro) > nro
    Mostrar doble " es el doble de " nro
  FinSi
FinAlgoritmo
```

Capítulo 9. MAYORES Y MENORES

9.1.- ¿Qué implica determinar el Mayor y Menor Valor?

El MAYOR o MENOR valor de un conjunto de elementos procesados mediante ciclos, son también salidas solicitadas. En este tipo de problemas, existen 3 variantes de requerimiento de información con respecto al mayor o el menor valor (Ver imagen).

Nótese que el 2do y el 3er requerimiento están determinados por el 1ero. Por ej.: la mayor o menor nota se debe conocer para determinar el nombre del alumno que la obtuvo.

(*) En los 2 últimos casos existe la posibilidad de que haya empates, es decir, que varios elementos obtengan el mismo valor mayor o menor. En este punto, sólo se podrá obtener el nombre del primero de ellos. Una de las formas más intuitivas de resolver estos casos es usar arreglos (se verá más adelante).



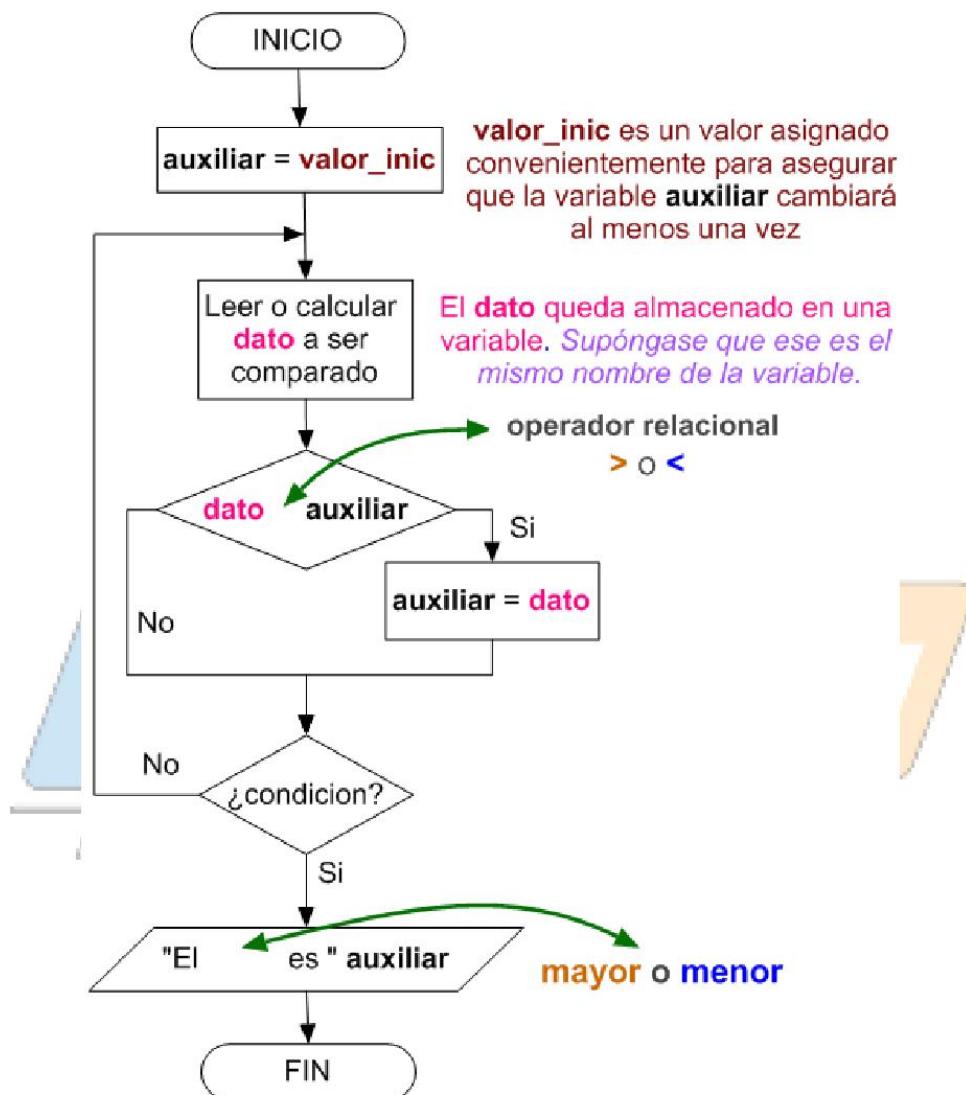
Para determinar el valor mayor/menor de los elementos procesados en un ciclo, ejemplificado aquí usando Repetir-Hasta.

A través de las iteraciones de un ciclo se debe comparar un "dato" (respecto al cual se quiere determinar el mayor o menor valor) con una variable "auxiliar" usando el operador relacional que corresponda y sólo en caso de cumplirse la condición, se debe actualizar el valor de la variable "auxiliar" con el valor que en ese momento cumple con ser mayor/menor.

Generalmente, el "dato" cambia de una iteración a otra.

IMPORTANTE: la variable "auxiliar" debe ser inicializada antes del ciclo, con un valor que asegure que ella cambiará al menos una vez.





Después de la primera comparación la variable "auxiliar" siempre almacenará el valor mayor/menor encontrado hasta el momento.

La variable "auxiliar" debe tener un nombre significativo, por ej.: si se estuviera

buscando la mayor y menor edad, dentro de un grupo de personas cuya edad se requerirá dentro del ciclo, las variables auxiliares respectivas se podrían identificar como: mayor_edad o menor_edad respectivamente.

Las eventualidades importantes a tener presente en este tipo de algoritmos, son las siguientes:

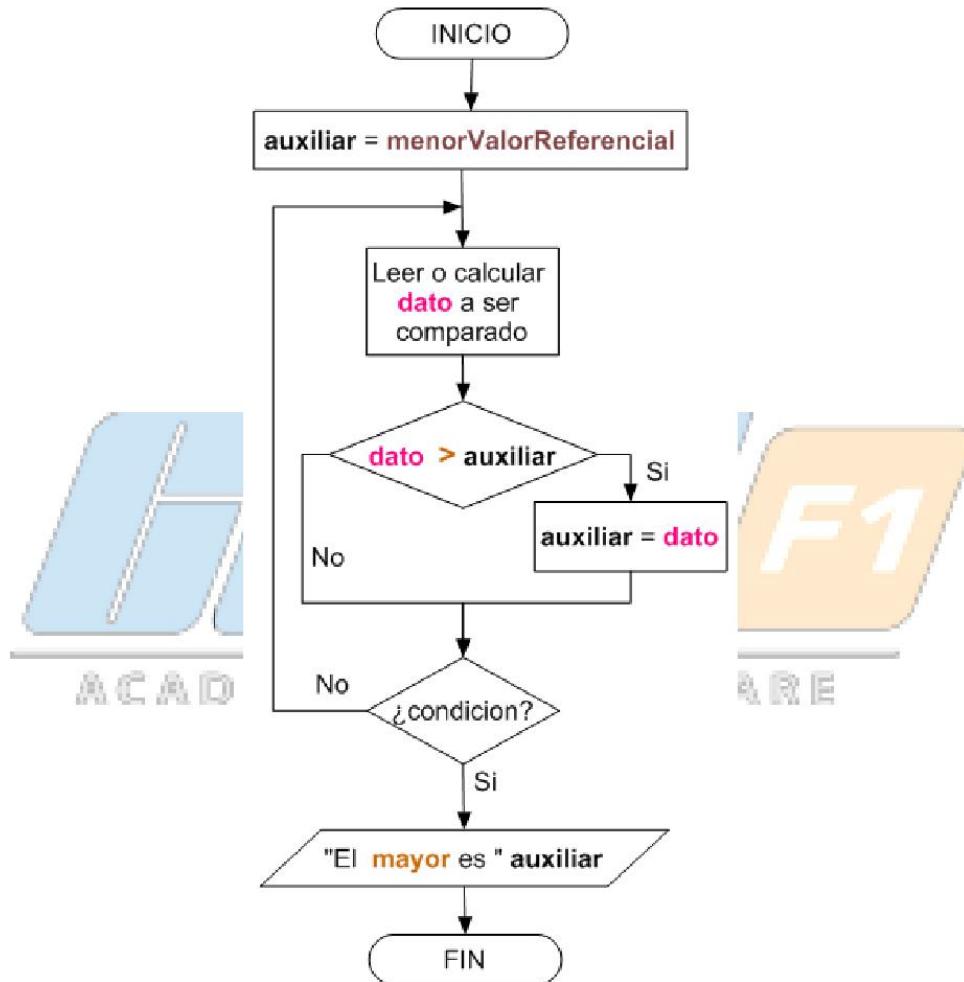
- 1) Los datos introducidos por el usuario en cada iteración del ciclo, no necesariamente se ingresarán de forma ordenada.
- 2) Se debe partir del hecho de que una variable leída o determinada que es objeto de comparación sólo puede tomar un valor a la vez.
- 3) Cada vez que el ciclo da una iteración, la variable leída (o determinada) toma un nuevo valor, lo que hace que se pierda el valor inmediato anterior. Esto conlleva, que al final del ciclo sólo se tenga el último valor que la variable tomó. Esta situación ocurre con cualquier tipo de ciclos y con cualquier tipo de variables.
- 4) Es requerido usar una variable "auxiliar" que permita saber en cualquier momento de la ejecución cual es el mayor o menor valor.
- 5) Se deben usar sentencias selectivas simples para evaluar el cumplimiento de la condición que permite "conocer la relación" entre el "dato" y "auxiliar".
- 6) La variable "auxiliar" se debe inicializar con un valor conveniente.

9.2.- Determinación de Mayores

En el caso de la determinación de mayores, la variable "auxiliar" debe ser inicializada con un valor muy bajo en relación con los valores que puede tomar la variable respecto a la cual se va a determinar el mayor valor. Por ej.: si se necesita determinar la mayor edad de un grupo de personas, la variable "auxiliar" debe inicializarse en 0, porque nadie tendrá una edad menor a 0.

Para la identificación del mayor valor en la condición a evaluar debe usarse el operador "mayor que".

En flujo gráfico usando un ciclo Repetir-Hasta el algoritmo es el siguiente:



El hecho de que la variable "auxiliar" sea inicializada con un valor muy bajo, asegura que en la primera iteración la condición se cumpla, lo cual implica que ella cambiará de valor al menos una vez.

También debe considerarse que si se desea determinar el mayor valor de una variable que puede tener valores negativos, dicha variable "auxiliar" no se puede inicializar en 0. Por ej.: si se necesita determinar la mayor temperatura de una ciudad que en ocasiones tiene temperaturas bajo cero, se debe inicializar "auxiliar" con un valor por debajo de la mínima temperatura alcanzada, por ejemplo, -10.

El siguiente es un ejemplo donde se leen las notas de 15 alumnos, con el objeto de determinar cuál fue la mayor nota entre las notas ingresadas:

```
Algoritmo Det_MayNota
    Definir nota,mayor_nota como Real
    Definir i Como Entero

    mayor_nota = -1
    Para i=1 hasta 15
        Mostrar "Ingrese nota del" i "º alumno: " Sin Saltar
        Leer nota
        Si nota > mayor_nota
            Entonces mayor_nota = nota
        FinSi
    FinPara
    Mostrar "La mayor nota fue " mayor_nota " ptos"
FinAlgoritmo
```

ACADEMIA DE SOFTWARE

Si se necesita mostrar algo que identifique al mayor valor (por ejemplo, un nombre), se debe utilizar una variable adicional. El siguiente es una extensión del ejemplo anterior, donde se mostrará el nombre del alumno (el 1ero para el que se cumple) que obtuvo la mayor nota:

```

Algoritmo Det_MayNota
    Definir nota,mayor_nota como Real
    Definir nomb_alumn,nomb_mayNota Como Caracter
    Definir i Como Entero

    mayor_nota = -1
    Para i=1 hasta 15
        Mostrar "Ingrese los datos del " i "º alumno"
        Mostrar "¿Nombre?" Sin Saltar
        Leer nomb_alumn
        Mostrar "¿Nota Obtenida?" Sin Saltar
        Leer nota
        Si nota > mayor_nota
            Entonces
                mayor_nota = nota
                nomb_mayNota = nomb_alumn
        FinSi
    FinPara
    Mostrar nomb_mayNota " obtuvo la mayor nota " Sin Saltar
    Mostrar "(" mayor_nota " ptos)"
FinAlgoritmo

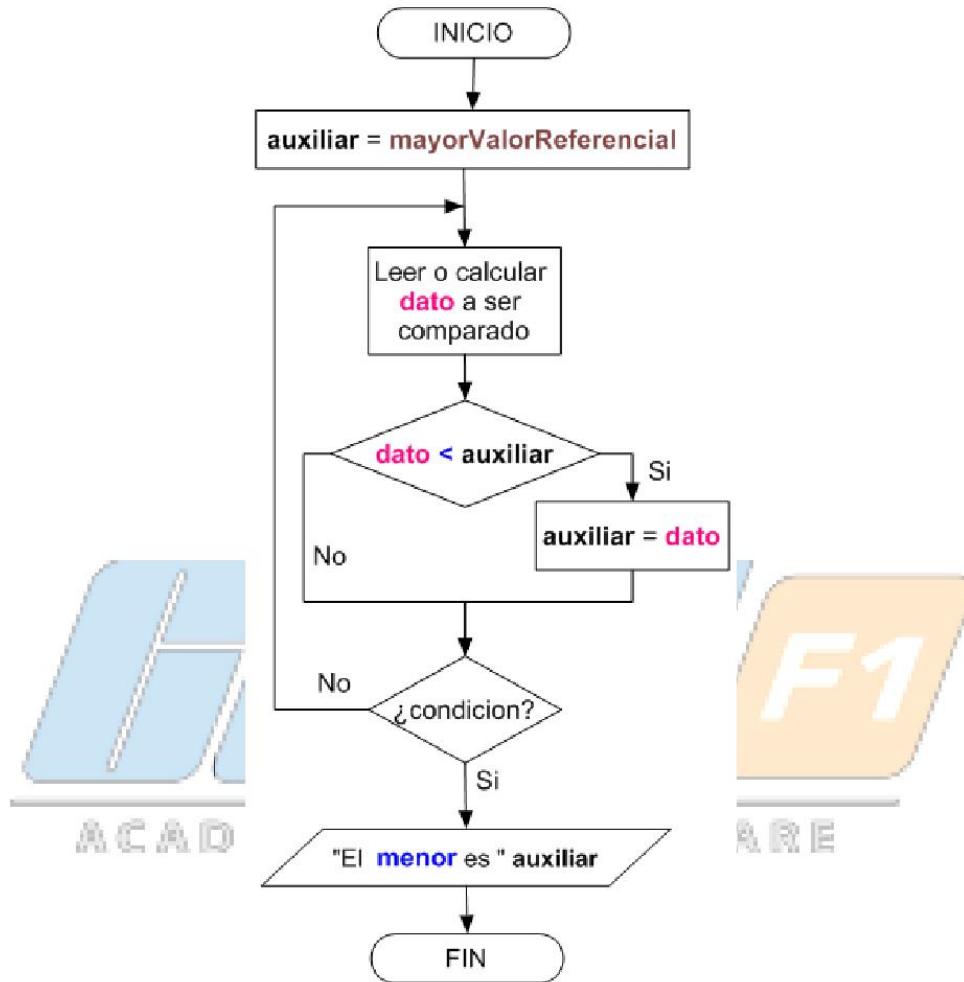
```

9.3.- Determinación de Menores

Para determinar el menor valor de un conjunto de valores, en la inicialización, a diferencia de cuando se determina el mayor, se debe inicializar la variable "auxiliar" en un valor muy alto, el cual debe ser un valor que supere ligeramente al valor más alto que pueda recibir la variable a evaluar.

El algoritmo para identificar el menor valor es casi igual al usado para la determinación del mayor valor, sólo es necesario usar el operador "menor que" en la condición.

El siguiente es el flujoGRAMA, usando un ciclo Repetir, para determinar el menor valor:



El hecho de que la variable "auxiliar" sea inicializada con un valor muy alto, asegura que en la primera iteración la condición se cumpla, lo cual implica que ella cambiará de valor al menos una vez.

Por ejemplo, si se requiere calcular la nota más baja de todos los alumnos, donde la nota puede tener valores enteros entre 0 y 100, la variable debe inicializarse en 101, debido a que es posible que la nota más baja sea 100 pts (en el caso de que todos los alumnos obtuvieran 100 pts), con la premisa de que nadie podrá obtener una nota más alta de 100 pts (asumiendo que los datos de entrada son válidos).

El siguiente es un ejemplo donde se va a leer la nota de 15 alumnos y necesita determinar cuál fue la menor nota entre las notas ingresadas:

```

Algoritmo Det_MenNota
    Definir nota,menor_nota como Real
    Definir i Como Entero

    menor_nota = 101
    Para i=1 hasta 15
        Mostrar "Ingrese nota del" i ° alumno: " Sin Saltar
        Leer nota
        Si nota < menor_nota
            Entonces menor_nota = nota
        FinSi
    FinPara
    Mostrar "La menor nota fue " menor_nota " ptos"
FinAlgoritmo

```

Si se necesita mostrar algo que identifique al menor valor (por ejemplo, un nombre), se debe utilizar una variable adicional. El siguiente es una extensión del ejemplo anterior, donde se mostrará el nombre del 1er alumno que obtuvo la menor nota:

```

Algoritmo Det_MenNota
    Definir nota,menor_nota como Real
    Definir nomb_alumn,nomb_menNota Como Caracter
    Definir i Como Entero

    menor_nota = 101
    Para i=1 hasta 15
        Mostrar "Ingrese los datos del" i ° alumno"
        Mostrar "¿Nombre?" Sin Saltar
        Leer nomb_alumn
        Mostrar "¿Nota Obtenida?" Sin Saltar
        Leer nota
        Si nota < menor_nota
            Entonces
                menor_nota = nota
                nomb_menNota = nomb_alumn
        FinSi
    FinPara
    Mostrar nomb_menNota " obtuvo la menor nota " Sin Saltar
    Mostrar "(" menor_nota " ptos)"
FinAlgoritmo

```

Capítulo 10. CONTADORES Y ACUMULADORES

10.1.- Generalidades Sobre Contadores y Acumuladores

Los Contadores y Acumuladores son variables que son muy útiles para distintos cálculos, por lo cual se debe conocer las premisas que a su respecto aplican:

- 1) Deben ser declaradas, para asociarle el tipo de dato que corresponda según el caso, para evitar la autodefinición de variables.
- 2) Deben ser inicializados antes del ciclo donde serán modificados según convenga. Esto tiene el propósito de evitar que la variable en cuestión finalice con un valor indeseado (basura [C++], NULL [php], undefined [JavaScript]).
- 3) La actualización o modificación de ambos tipos de variable se debe colocar dentro de los ciclos (en cualquiera de sus variantes).
- 4) Generalmente, el valor final de ellas se utilizará fuera de los ciclos donde son modificadas.
- 5) Es posible utilizar sus valores parciales para proveer información al usuario dentro del ciclo.

10.2.- Concepto de Contador

Un CONTADOR es una variable numérica entera, cuyo valor se va incrementando de uno en uno. El objetivo principal de un contador, es como su nombre lo indica, llevar la cuenta de la ocurrencia de algo durante la ejecución del algoritmo o programa.

Por ejemplo: Si se van a leer datos de alumnos como: nombre, sexo y edad, se pueden llevar los siguientes contadores:

- contador de mujeres,
- contador de hombres,
- contador de personas mayores de 25 años,
- contador de mujeres menores de edad, etc.

La sintaxis general para actualizar un contador es la siguiente:

```
contador = contador + 1
```

Un contador puede incrementarse o no según el caso. Ej:

```
contador1 = 0      // Inicialización del contador1
contador2 = 0      // Inicialización del contador2

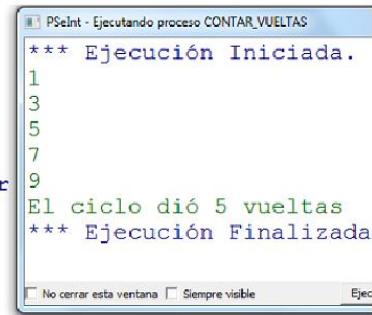
Para i = 1 hasta 10
    contador1 = contador1 + 1          // Se incrementa siempre
    Si < condición >
        Entonces contador2 = contador2 + 1 // No se incrementa siempre
    FinSi
FinPara
```

Mostrar 'El contador 1 dió:' contador1
 Mostrar 'El contador 2 dió:' contador2

10.3.- Contador General y Específico

Un CONTADOR GENERAL se incrementa en todas las iteraciones del bucle, su variación no depende de ninguna circunstancia. En el ejemplo se aprecia la declaración, inicialización, incremento y mostrado de la variable contador:

```
1 Algoritmo Contar_Vueltas
2     Definir i Como Entero
3     Definir contador Como Entero
4
5     contador=0 // se inicializa
6
7     Para i=1 hasta 10 con paso 2 Hacer
8         Mostrar i
9         // se actualiza
10        contador=contador+1
11    FinPara
12    // se utiliza
13    Mostrar "El ciclo dió " contador " vueltas"
14 FinAlgoritmo
```



```
*** Ejecución Iniciada.
1
3
5
7
9
El ciclo dió 5 vueltas
*** Ejecución Finalizada
```

Un CONTADOR ESPECÍFICO es aquél que se incrementa sólo cuando se cumple algo. Esto implica, que se debe evaluar una condición (simple o compuesta) antes de actualizar el contador.

En el siguiente algoritmo se evalúan 10 números, bajo la premisa de que el cero (0) no será ingresado, y se manejan 2 contadores específicos:

- positivos : se incrementará cuando un valor sea positivo.
- negativos: otro cuando el número sea negativo.

¿Qué habría que añadir si se quisiera manejar la posibilidad de que el usuario ingrese el 0?

```
1  Algoritmo Pos_Y_Neg
2      Definir num,i Como Entero
3      Definir positivos,negativos Como Entero
4
5      positivos=0
6      negativos=0
7      Para i=1 hasta 10 Hacer
8          Mostrar "Ingrese un número entero:"
9          Ler num
10         Si num>0
11             entonces positivos=positivos+1
12             sino negativos=negativos+1
13         FinSi
14     FinPara
15     Mostrar "Positivos: " positivos
16     Mostrar "Negativos: " negativos
17 FinAlgoritmo
```

En este otro ejemplo, se trabaja también con contadores específicos. En él se leen 10 números y se determina cuantos de ellos son pares y cuantos son impares:

```

1 Algoritmo Par_Y_Impar
2     Definir num,i Como Entero
3     Definir pares,impares Como Entero
4
5     pares=0
6     impares=0
7     Para i=1 hasta 10 Hacer
8         Mostrar "Ingrese un número entero:"
9         Leer num
10        Si (num mod 2) = 0
11            entonces pares=pares+1
12            sino impares=impares+1
13        FinSi
14    FinPara
15    Mostrar "Pares: " pares
16    Mostrar "Impares: " impares
17 FinAlgoritmo

```

10.4.- Concepto de Acumulador

Un ACUMULADOR es una variable numérica entera o real, cuyo valor podría irse incrementando (o decrementando) en un valor variable. Se suele utilizar éste tipo de variable para llevar un acumulado de valores obtenidos en el algoritmo o programa.

Por ejemplo, si se van a leer datos de alumnos como nombre, sexo y nota, se pueden determinar los siguientes acumuladores:

- acumulador de todas las notas,
- acumulador de notas de las mujeres,
- acumulador de notas de los hombres,
- acumulado de notas de un alumno, etc.

La sintaxis general para actualizar un acumulador es la siguiente:

$$\text{acumulador} = \text{acumulador} + \text{variable}^*$$

(*) variable contiene el valor que se necesita acumular. Normalmente ésta es un dato que se lee (dato de entrada) o se calcula (dato de proceso) dentro del ciclo donde se modifica la variable.

Un acumulador puede incrementarse o no según el caso. Ej:

```
acumulador1 = 0 // Inicialización del acumulador1
acumulador2 = 0 // Inicialización del acumulador2

Para i = 1 hasta 10
    acumulador1 = acumulador1 + variable // Se incrementa siempre
    Si < condición >
        Entonces acumulador2 = acumulador2 + variable // No se incrementa siempre
    FinSi
FinPara
```

Mostrar 'El acumulador 1 dió: ' acumulador1
 Mostrar 'El acumulador 2 dió: ' acumulador2

10.5.- Acumulador General y Específico

Un ACUMULADOR GENERAL se incrementa en todas las iteraciones del bucle, su variación no depende de ninguna circunstancia. En el ejemplo se muestra cómo acumular el sueldo (se maneja como premisa que ese dato siempre será positivo) de 100 empleados:

```
1 Algoritmo Acumular_Sueldos
2   Definir i Como Entero
3   Definir acum_sueldos,sueldo como Entero
4   Definir nombre Como Caracter
5
6   acum_sueldos=0
7   Para i=1 hasta 100 Hacer
8     Limpiar Pantalla
9     Mostrar "Ingrese el nombre y el sueldo del " i "º empleado"
10    Leer nombre,sueldo
11    Mostrar "El sueldo de " nombre " incrementará el acumulado en : " sueldo
12    Esperar 2 Segundos
13    acum_sueldos=acum_sueldos+sueldo
14  FinPara
15  Mostrar "El acumulado de todos los sueldos dió: " acum_sueldos
16 FinAlgoritmo
```

Un ACUMULADOR ESPECÍFICO es aquél que se incrementa sólo cuando se cumple algún requisito. Esto implica, que se debe evaluar una condición previamente a la actualización del mismo. En el siguiente ejemplo, se usa una selectiva de alternativa múltiple antes de acumular.

```
1  Algoritmo Acumular_Sueidos
2      Definir i Como Entero
3      Definir acum_sueldos_hom,acum_sueldos_muj,sueldo como Real
4      Definir nombre,sexo Como Caracter
5
6      acum_sueldos_hom=0
7      acum_sueldos_muj=0
8      Para i=1 hasta 100 Hacer
9          Limpiar Pantalla
10         Mostrar "Ingrese el nombre, sexo (F/M), y el sueldo del " i "º empleado"
11         Leer nombre,sexo,sueldo
12         Segun Mayusculas(Sexo)
13             "F": acum_sueldos_muj=acum_sueldos_muj+sueldo
14             "M": acum_sueldos_hom=acum_sueldos_hom+sueldo
15         FinSegun
16         Mostrar "El sueldo de " nombre " incrementará el acumulador respectivo en: " sueldo
17         Esperar 2 Segundos
18     FinPara
19     Mostrar "El acumulado de sueldos de las mujeres dió: " acum_sueldos_muj
20     Mostrar "El acumulado de sueldos de los hombres dió: " acum_sueldos_hom
21 FinAlgoritmo
```



ACADEMIA DE SOFTWARE

Capítulo 11. PROMEDIOS Y PORCENTAJES 2

11.1.- Introducción a Cálculos Usando Ciclos

En este capítulo, se verá como calcular promedios y porcentajes que dependen de variables tipo contador y acumulador que son modificadas usando ciclos, y por ello es muy relevante tener presente los siguientes lineamientos:

- 1) Todas las variables implicadas deben ser declaradas.
- 2) Los contadores y acumuladores deben ser inicializados fuera del ciclo dentro del cual serán modificad@s, según convenga a la obtención de los resultados requeridos.
- 3) Las variables tipo contador o tipo acumulador deben ser incrementados o actualizados dentro del ciclo que convenga.
- 4) Los promedios y porcentajes deben ser calculados fuera de el(los) ciclo(s) donde se han modificado las variables de tipo contador y/o acumulador implicadas en el cálculo de las primeras.
- 5) Se deben evitar las divisiones entre 0, haciendo las verificaciones a las que haya lugar.

11.2.- Promedio

Los promedios son cálculos muy frecuentemente solicitados en los problemas donde hay ciclos (de cualquier tipo). Por ello, son típicos requerimientos tales como "Calcule el promedio de..."

- notas de una sección,
- montos pagados por los clientes,
- edades de las personas cuyos datos se han procesado, etc.

El cálculo de un promedio se fundamenta en acumular y contar, para luego plantear una división entre ellos. Es por eso que la forma general para calcular un promedio es:

$$\text{promedio} = \text{acumuladorAlgo} / \text{contadorAlgo}$$

Pudiera darse el caso de que el valor del acumulador sea 0, después de ejecutar las iteraciones del ciclo, como consecuencia el promedio resultaría en 0 (el resultado de dividir 0 entre cualquier número da 0).

El contador implicado NO debe ser 0, porque la división entre 0 no existe.

El siguiente es algoritmo sirve para calcular y mostrar de un promedio de 10 números, empleando un ciclo que tiene una cantidad fija de iteraciones. En este caso, la división se hace entre 10, porque es el número de datos procesados, constituyéndose éste en un contador cuyo valor máximo se conoce de antemano:

```

1  Proceso Promediar_Nums
2    Definir i Como Entero
3    Definir num,suma,promedio Como Real
4
5    Para i=1 hasta 10 Hacer
6      Mostrar "Ingrese el " i "º número" Sin Saltar
7      Leer num
8
9      suma= suma + num // incremento del acumulador
10   FinPara
11
12   promedio= suma / 10 // cálculo del promedio
13   Mostrar "El promedio de los 10 números fue: " promedio
14   FinProceso
```

Si el número de vueltas del ciclo es variable, por ej., porque el bucle se rompe cuando el usuario responde una interrogante o selecciona la opción "salir" (en el caso de los menús), se debe emplear una variable contador que será el divisor del acumulador.

En el ejemplo se calcula el promedio de sueldo de una cantidad desconocida de empleados.

IMPORTANTE: En este caso no hay posibilidad de que el contador sea 0, porque se está usando un ciclo Repetir.

```

Algoritmo Prom_Sueldos
    Definir sueldo,acum_sueldos,prom_suel Como Real
    Definir cont_emp Como Entero
    Definir resp Como Caracter
    acum_sueldos = 0 // Inicialización del acumulador general
    cont_emp = 0 // Inicialización del contador general
    // Ciclo donde se modifican las variables determinantes
    // para el cálculo del promedio general
    Repetir
        // Incremento del contador
        cont_emp = cont_emp + 1
        Mostrar "Ingrese el " cont_emp " sueldo" Sin Saltar
        Leer sueldo
        // Incremento del acumulador
        acum_sueldos = acum_sueldos + sueldo
        Mostrar "¿Desea continuar S/N?" Sin Saltar
        Leer resp
        resp = Mayusculas(resp)
    Mientras Que (resp <> "N") // Fin del ciclo
        // Cálculo y despliegue del promedio de sueldos
        prom_suel = acum_sueldos / cont_emp
        Mostrar "El promedio de los sueldos es:" prom_suel
    FinAlgoritmo

```



Al igual que con los contadores específicos, en ocasiones se necesita calcular el promedio de un conjunto de valores que dependen de que se cumpla algo.

En esos casos se debe tener especial cuidado, debido a que es posible que el contador nunca se incremente, lo que podría generar una división entre 0.

Véase el ejemplo, en el que se necesita obtener el promedio de precios de los pantalones modelo 1, considerando que pudieran tener precios diferentes según la talla:

```

Algoritmo Prom_Prec_PantM1
    Definir prec,acum_prec_PT1,prom_prec_PM1 como Real
    Definir mod_pant,cont_PM1 Como Entero
    Definir resp Como Caracter
    cont_PM1 = 0 // Inicialización del contador específico
    acum_prec_PM1 = 0 // Inicialización del acumulador específico
    // Ciclo donde se modifican las variables determinantes
    // para el cálculo del promedio específico
    Repetir
        Mostrar "Ingrese el modelo del pantalón (1/2/3)" Sin Saltar
        Leer mod_pant
        Mostrar "Ingrese el precio unitario" Sin Saltar
        Leer prec
        // Hacer la evaluación necesaria para incrementar
        // las variables específicas
        Si mod_pant = 1 Entonces
            cont_PM1 = cont_PM1 + 1
            acum_prec_PM1 = acum_prec_PM1 + prec
        FinSi
        Mostrar "¿Desea continuar S/N?" Sin Saltar
        Leer resp
        Mientras Que (Mayusculas(resp) <> "N") // Fin del ciclo
        Si cont_PM1 > 0 // Evaluación del denominador
            Entonces prom_prec_PM1 = acum_prec_PM1 / cont_PM1
            Sino prom_prec_PM1 = 0
        FinSi
        Mostrar "Promedio de precios de los " Sin Saltar
        Mostrar " pantalones modelo 1: " prom_prec_PM1 " Bs."
    FinAlgoritmo

```

ACADEMIA DE SOFTWARE

11.3.- Porcentajes

Los porcentajes son similares a los promedios en cuanto a que su cálculo depende de variables que se van modificando (o no) iterativamente. Es común que sea requerido "calcular el porcentaje de...: aprobados, clientes que compró más de 3 artículos, menores de edad, etc.

El cálculo de un porcentaje se fundamenta en determinar la proporcionalidad que hay entre un contador específico y un contador general. Po ello, la fórmula general para su cálculo es:

$$\text{porcentaje} = \text{contadorEspecifico} / \text{contadorGeneral} * 100$$

Los contadores se actualizan dentro una estructura repetitiva y el porcentaje se calcula luego de finalizar las iteraciones. En caso de que el "contadorEspecifico" sea 0, el porcentaje resultaría 0 (0 entre cualquier número resulta 0), pero si el "contadorGeneral" es 0, no se debe realizar la división, porque ocurriría un error de ejecución, debido a que la división entre 0 no existe.

Los porcentajes calculados según se refiere, siempre involucran el uso de al menos un contador específico, pero pudiera implicar 2 de ellos, es decir, el numerador pudiera ser más específico que el denominador.

Por ejemplo, si necesita calcular el porcentaje de aprobados de una sección de 15 alumnos, se debe primero llevar un contador de todos los alumnos aprobados, el cual se va incrementando dentro del ciclo y luego se divide entre la cantidad de alumnos. El algoritmo de este ejemplo es el siguiente:

```


Algoritmo CalcPorAprob
    Definir cont_aprob,i Como Entero
    Definir porc_aprob Como Real

    cont_aprob = 0 // Inicialización del contador específico
    Mostrar "Ingrese c/u de las 15 notas en el rango de 0 a 20"
    Para i=1 Hasta 15
        Mostrar "¿Nota " i "/15?" Sin Saltar
        Leer nota
        Si nota > 10 Entonces
            // Incremento del contador específico
            cont_aprob = cont_aprob + 1
        FinSi
    FinPara
    porc_aprob = cont_aprob/15*100 // Cálculo del porcentaje
    Mostrar "El porcentaje de aprobados fue " porc_aprob "%"
FinAlgoritmo

```

En ocasiones se necesitan calcular porcentajes de contadores específicos. Por ejemplo, el porcentaje de aprobados de los hombres. En este caso, existen 2 contadores específicos implicados en el cálculo.

En este tipo de casos se debe tener la precaución de no hacer una división entre 0:

```
Algoritmo Porc_HombAprob
    Definir cont_hom_aprob,cont_homb,nota,porc_homb_aprob Como Real
    Definir i como Entero
    Definir genero como Caracter

    cont_homb = 0 // Inicialización de contador MENOS específico
    cont_hom_aprob=0 // Inicializ. de contador MÁS específico
    Para i=1 hasta 20 Hacer
        Mostrar "Ingrese el género (F/M)" Sin Saltar
        Leer genero
        Mostrar "Ingrese la nota obtenida (0-20)" Sin Saltar
        Leer nota
        Si genero = "M" Entonces
            // Incremento del contador MENOS específico
            cont_homb = cont_homb + 1
            Si nota > 10 Entonces
                // Incremento del contador MÁS específico
                cont_hom_aprob = cont_hom_aprob + 1
            FinSi
        FinSi
    FinPara
    // Se verifica si el cont_homb es mayor que 0
    Si cont_homb > 0 Entonces
        porc_homb_aprob = cont_hom_aprob / cont_homb * 100
    Sino
        porc_homb_aprob=0
    FinSi
    Mostrar "Porcentaje de hombres aprobados : " porc_homb_aprob "%"
FinAlgoritmo
```

Capítulo 12. ESTRUCTURAS REPETITIVAS DOBLES. PARTE 1

12.1.- Estructura Repetitiva Anidada

Al igual que se pueden colocar unas selectivas dentro de las otras, los bucles pueden estar unos dentro de otros. Al anidar bucles, hay que tener en cuenta que el bucle interno funciona como una sentencia más en el bloque del bucle externo, por lo tanto, en cada iteración del bucle externo se van a ejecutar todas las iteraciones del bucle interno.

Los bucles deben estar bien formados y ser sintácticamente correctos, para que pueda haber un anidamiento válido, en otras palabras, nunca pueden cruzarse las sentencias de los bucles. Si se tiene un ciclo "Para" (también llamado "Desde") que internamente tiene un ciclo "Repetir", es necesario finalizar las sentencias o instrucciones del ciclo más interno, en este caso, el "Repetir" antes de colocar la culminación del ciclo externo "Para".

Si el bucle externo se repite M veces y el interno se repite N veces, y por cada iteración del externo se repite el interno, entonces el número total de iteraciones será el producto de M y N.

Los bucles que se anidan pueden ser de igual o distinto tipo, por lo tanto, cualquier combinación es válida.

ACADEMIA DE SOFTWARE

Se puede hacer anidamiento de 2, 3, 4 o más ciclos.

El siguiente ejemplo de anidamiento de bucles "Para" sirve para la lectura de los nombres de las asignaturas, en un centro educativo de idiomas donde hay 8 profesores y cada uno tiene 5 asignaturas:

```

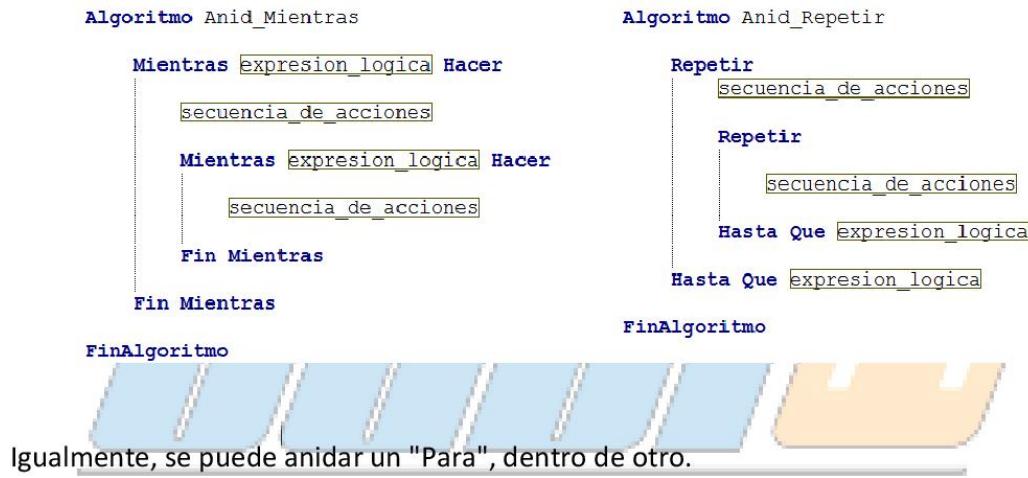
Para i = 1  hasta 8
  Para k = 1  hasta 5
    Escribir "Profesor" i "introduzca su asignatura N°" k
    Leer asignatura
  FinPara
FinPara

```

12.2.- Combinación de Ciclos Anidados

Cómo se indicó los ciclos se pueden anidar usando cualquier combinación, incluyendo la posibilidad que en un anidamiento de 2 ciclos, ambos sean del mismo tipo:

- "Mientras-Hacer" externo y "Mientras-Hacer" interno.
- "Repetir-Hasta" externo y "Repetir-Hasta" interno.



ACADEMIA DE SOFTWARE

```

Algoritmo Anid_Para
// La variable numérica de un PARA debe ser diferente a la del otro
    Para variable_numerica<-valor_inicial Hasta valor_final Con Paso paso Hacer
        secuencia_de_acciones
        Para variable_numerica<-valor_inicial Hasta valor_final Con Paso paso Hacer
            secuencia_de_acciones
        Fin Para
    Fin Para
FinAlgoritmo

```

El siguiente ejemplo muestra como el anidamiento de dos (2) ciclos "Para" genera en pantalla el despliegue de las tablas de multiplicar desde 1 hasta 10:

```

1 Algoritmo Tabla_Multip
2     Definir i,j Como Entero
3
4     Para i=1 hasta 10 Hacer
5         Mostrar "TABLA del #" i;
6
7             Para j=0 hasta 10 Hacer
8
9                 Escribir i " * " j " = " i*j
10            FinPara
11
12        FinPara
13 FinAlgoritmo

```

En el siguiente ejemplo también se generan las tablas de multiplicar del 1 al 10, usando un ciclo "Mientras-Hacer" (lo cual requiere ir incrementando la i manualmente) como ciclo externo, y un ciclo "Para" como ciclo interno:

1 Algoritmo Tabla_Multip
2 Definir i,j Como Entero
3
4 i=1
5 Mientras i<=10 Hacer
6 Mostrar "TABLA del #" i;
7
8 Para j=0 hasta 10 Hacer
9
10 Escribir i " * " j " = " i*j
11 FinPara
12
13 i=i+1
14
15 FinMientras
16 FinAlgoritmo

12.3.- Lugar Para Entradas y Salidas

Cuando se van a anidar sentencias para obtener entradas y/o mostrar salidas (así como para sentencias de proceso) es muy importante identificar donde se van a colocar las instrucciones respectivas. En el siguiente ejemplo:

```

Algoritmo ProfYNotas
    // Definición de variables
    Mostrar "¿Nombre del profesor?" Sin Saltar
    Leer nom_prof
    Mostrar "¿Cantidad de estudiantes?" Sin Saltar
    Leer cant_est

    Para i=1 hasta cant_est
        Limpiar Pantalla
        Mostrar "¿Nombre del estudiante?" Sin Saltar
        Leer nomb_est
        Mostrar "Cant. de notas a procesar?" Sin Saltar
        Leer cant_not_est

        Para j=1 hasta cant_not_est
            Mostrar "¿Nota " j "?" Sin Saltar
            Leer nota
            Si nota>=0 y nota<=100
                entonces Mostrar "Nota válida"
                sino Mostrar "Nota no válida"
            FinSi
        FinPara

        Mostrar "Al estudiante " nomb_est Sin Saltar
        Mostrar " se le procesaron " cant_not_est " notas"
        Mostrar ""
        Mostrar "Pulse cualquier tecla para proseguir"
        Esperar tecla
    FinPara

    Mostrar "El profesor " nom_prof " proceso " Sin Saltar
    Mostrar notas " de " cant_est " estudiantes."
FinAlgoritmo

```

Se leen entradas: antes del ciclo externo y dentro de él y dentro del ciclo interno.
 Se muestran salidas dentro del ciclo interno, dentro del ciclo externo y fuera de él.

Capítulo 13. ESTRUCTURAS REPETITIVAS DOBLES. PARTE 2

13.1.- Contadores y Acumuladores en doble Ciclos

Ante la necesidad de utilizar contadores y acumuladores en algoritmos con ciclos anidados, se debe tener mucho cuidado con lo siguiente:

- donde se deben inicializar y
- donde se deben actualizar

Errar en este sentido, generará un error lógico y por ende un resultado diferente al esperado.

En el siguiente ejemplo se realiza el conteo de notas en las que todos los alumnos obtuvieron 100 puntos, y por c/alumno se contabilizan los ceros que obtuvo y el acumulado de sus notas.



```

Algoritmo ProfYNotas_ContYAcum
    // Definición de variables
    Mostrar "¿Nombre del profesor?" Sin Saltar
    Leer nom_prof
    Mostrar "¿Cantidad de estudiantes?" Sin Saltar
    Leer cant_est
    cont_100_secc=0 // contador de 100 obtenidos en la secc
    Para i=1 hasta cant_est
        Limpiar Pantalla
        Mostrar "¿Nombre del estudiante y cantidad de notas obtenidas?"
        Leer nomb_est,cant_not_est

        cont_0_est=0 // contador de 0 de un estudiante
        acum_not_est=0 // acumulador de notas de un estud.
        Para j=1 hasta cant_not_est
            Mostrar "¿Nota " j "?" Sin Saltar
            Leer nota
            Si nota=0
                entonces cont_0_est=cont_0_est+1 // Act. Cont. x Estudiante
            FinSi
            Si nota=100
                entonces cont_100_secc=cont_100_secc+1 // Act. Cont X Sección
            FinSi
            acum_not_est=acum_not_est+nota // Act. Acum. X Estudiante
        FinPara
        Mostrar nomb_est " tiene " cant_not_est " resultando con: "
        Mostrar cont_0_est " evaluaciones con 0 ptos."
        Mostrar "y un acumulado de notas de " acum_not_est " ptos."

    FinPara
    Mostrar "En la sección del profesor " nom_prof Sin Saltar
    Mostrar "los estudiantes obtuvieron 100 " cont_100_secc " veces"
FinAlgoritmo

```

13.2.- Promedios y Porcentajes en doble Ciclos

Usando anidamiento de ciclos también es común calcular promedios y porcentajes. Para ello es fundamental:

- inicializar las variables donde convenga,
- actualizar los acumular donde sea pertinente, y
- hacer el cálculo de promedio o porcentaje fuera de ciclo que corresponda.

El siguiente ejemplo se calculan 2 tipos de promedio:

- 1) Promedio de notas de cada est.: el acumulador se inicializa antes del interno interno (línea 12), se acumula dentro ese ciclo (línea 16) y se calcula fuera de él (línea 18).

- 2) Promedio de notas de la sección: el acumulador se inicializa antes del ciclo externo (línea 7), se acumula luego del ciclo interno (línea 21) y se calcula al salir del ciclo externo (línea 23).

```

1  Algoritmo ProfYNotas_PromNotas
2      // Definición de variables
3      Mostrar "¿Nombre del profesor?" Sin Saltar
4      Leer nom_prof
5      Mostrar "¿Cantidad de estudiantes?" Sin Saltar
6      Leer cant_est
7      acum_not_secc=0 // Inicializ. Acum. de Notas de la Sección
8      Para i=1 hasta cant_est
9          Limpiar Pantalla
10         Mostrar "¿Nombre del estudiante y cantidad de notas obtenidas?"
11         Leer nomb_est,cant_not_est
12         acum_not_est=0 // Inicializ. Acum. de Notas de un Estudiante
13         Para j=1 hasta cant_not_est
14             Mostrar "¿Nota " j "?" Sin Saltar
15             Leer nota
16             acum_not_est=acum_not_est+nota // Act. Acum. X Estudiante
17         FinPara
18         prom_est=acum_not_est/cant_not_est // Cálculo Prom. X Estudiante
19         Mostrar nomb_est " obtuvo tiene " cant_not_est " notas " Sin Saltar
20         Mostrar "y obtuvo un promedio de " prom_est " ptos."
21         acum_not_secc=acum_not_secc+acum_not_est // Act. Acum. Notas de Sección
22     FinPara
23     prom_secc=acum_not_secc/cant_est // Calc. Prom. Sección
24     Mostrar "En la sección del profesor " nom_prof Sin Saltar
25     Mostrar "el promedio general de notas fue: " prom_secc
26 FinAlgoritmo

```

ACADEMIA DE SOFTWARE

El siguiente ejemplo se calculan 2 tipos de porcentaje:

- 1) Porc. de notas perfectas de un estudiante, para lo cual, el contador se inicializa en la línea 13 y se actualiza en la línea 18. Para luego calcular el porc. en la línea 23.
- 2) Porc. aprobados de la sección, el contador se inicializa en la línea 7, se actualiza en la línea 27. Para luego calcular el porc. en la línea 30.

```

1  Algoritmo ProyNotas_PromNotas
2      // Definición de variables
3      Mostrar "¿Nombre del profesor?" Sin Saltar
4      Leer nom_prof
5      Mostrar "¿Cantidad de estudiantes?" Sin Saltar
6      Leer cant_est
7      cont_aprob_secc=0 // Inicializ. Cont. de Aprob. de la Sección
8      Para i=1 hasta cant_est
9          Limpiar Pantalla
10         Mostrar "¿Nombre del estudiante y cantidad de notas obtenidas?"
11         Leer nomb_est,cant_not_est
12         acum_not_est=0 // Inicializ. Acum. de Notas de un Estudiante
13         cont_not100_est=0 // Inicializ. Cont. de nota perfecta de un Estudiante
14         Para j=1 hasta cant_not_est
15             Mostrar "¿Nota " j "?" Sin Saltar
16             Leer nota
17             Si nota=100 entonces
18                 cont_not100_est=cont_not100_est+1 // Act. Cont de 100 X Estudiante
19             FinSi
20             acum_not_est=0 // Inicializ. Acum. de Notas de un Estudiante
21         FinPara
22         prom_est=acum_not_est/cant_not_est // Cálculo Prom. X Estudiante
23         porc_not100_est=cont_not100_est/cant_not_est // Cálculo Prom. X Estudiante
24         Mostrar nomb_est " obtuvo tiene " cant_not_est " notas " Sin Saltar
25         Mostrar "y su porcentaje de notas perfectas fue " porc_not100_est "%"
26         Si prom_est>=50 entonces
27             cont_aprob_secc=cont_aprob_secc+1 // Act. Cont. Aprob. de la Sección
28         FinSi
29     FinPara
30     porc_aprob_secc= cont_aprob_secc/cant_est // Calc. Porc. de Aprob de la Sección
31     Mostrar "En la sección del profesor " nom_prof Sin Saltar
32     Mostrar "el porcentaje de aprobados fue: " porc_aprob_secc "%"
33 FinAlgoritmo

```

ACADEMIA DE SOFTWARE

Capítulo 14. BÚSQUEDA EN ARREGLOS

14.1.- Generalidades de la Búsqueda en Arreglos

Buscar un elemento en un arreglo consiste en determinar si un valor se encuentra en alguna de las posiciones del arreglo. La búsqueda puede ser de un elemento único o de un elemento que puede repetirse (no único). Para buscar un elemento en un arreglo se debe usar un ciclo y se debe revisar cada posición del arreglo con una instrucción condicional simple, con el objeto de comparar cada elemento del arreglo con el valor que se está buscando. En forma general, el algoritmo de búsqueda es el siguiente:

```
Dimension arreglo[tam_arreglo]
encontrado=falso
Para posic=primera_posic hasta ultima_posic
    Si arreglo[posic] = elementoBuscado
        entonces encontrado=verdadero
    FinSi
FinPara
```

"Encontrado" es una variable lógica que se utiliza para "saber" si se encontró el elemento buscado o no. Según la indexación ultima_posic tiene un valor diferente:

En el algoritmo que se acaba de ver, es importante destacar cual es el valor de primera_posic y de ultima_posic dependiendo del tipo de indexación:

Si se está usando indexación en base 1:

- primera_posic es 1
- ultima_posic es tam_arreglo

Si se está usando indexación en base 0:

- primera_posic es 0
- ultima_posic es tam_arreglo-1

Como se ha venido usando desde el nivel anterior, los ejemplos se trabajan con arreglos con indexación en base 1.

14.2.- Buscar un Elemento Único

Cuando la búsqueda es de valores únicos, NO se debe utilizar un ciclo "Para", ya que en este tipo de casos es muy posible que no sea necesario recorrer todo el arreglo hasta encontrar el elemento buscado, porque no se sabe si dicho elemento se va a encontrar en la 1era posición, en la última posición o en algún lugar intermedio. Es por esto, que debe usarse un ciclo "Repetir" o un "Mientras", ya que si se usara un "Para" o un "Para Cada" el algoritmo no sería del todo eficiente.

Los siguientes son ejemplos de búsqueda de un elemento que debe ser único, como un número de cédula de identidad en un arreglo llamado cedulas donde se encuentran almacenadas los números de cédulas de N personas.

En la parte superior, se aprecia la implementación sin usar modularidad y en la parte inferior se presenta una función aplicando modularidad:



----- Funcionalidad de búsqueda de cédula sin usar modularidad -----

```
// Carga del arreglo de cédulas
encontrado=False
i=1 // i se usará como índice en el arreglo
Repetir
    Si cedulas[i]=="12345678"
        entonces encontrado=True
        sino i=i+1
    FinSi
Hasta Que (encontrado=True) o (i>N)
// N es el tamaño del arreglo con indexación base-1
```

----- Módulo implementado como Función para Buscar una cédula -----

```
Funcion encontrado <-Func_Busc_Ced(N,cedulas)
    Definir i como entero
    encontrado=False
    i=1 // i se usará como índice en el arreglo
    Repetir
        Si cedulas[i]=cedbuscada
            entonces encontrado=True
            sino i=i+1
        FinSi
    Hasta Que (encontrado=True) o (i>N)
FinFuncion
```

ACADEMIA DE SOFTWARE

14.3.- Buscar un Elemento Repetido

Con arreglos, se puede informar si hay varios elementos iguales dentro de un arreglo, lo cual era imposible antes de trabajar con arreglos.

Cuando se recorre un arreglo en búsqueda de elementos y existe la posibilidad de que ellos se repitan, se puede usar cualquiera de los ciclos vistos, porque el arreglo debe recorrerse de principio a fin.

Este es un caso de búsqueda de valores no únicos. Por ejemplo, usando un arreglo que tiene almacenados los modelos de los equipos vendidos en una tienda de celulares, se pueden saber cuantas veces se fue vendido un modelo particular. El siguiente es la porción del algoritmo donde se emplea el ciclo "Paraca Cada" con el fin de llevar adelante dicho conteo:

```
Algoritmo Busq_Modelos
    Dimension modelos_equipo[N]
    // Carga del arreglo de los modelos de equipos vendidos
    Mostrar "Introduzca el código de modelo a buscar" Sin Saltar
    Leer cod_modelo
    cant_mod_busc=0 // cantidad del modelo buscado
    Para cada modelo de modelos_equipos
        Si modelos_equipo[i]=cod_modelo
            entonces cant_mod_busc=cant_mod_busc+1
        FinSi
    FinPara
    Mostrar "El modelo indicado se encontró " cant_mod_busc " veces"
FinAlgoritmo
```



Capítulo 15. PROCESANDO VALORES EN ARREGLOS

15.1.- Promedio Usando Arreglos

Para calcular el promedio de los valores guardados en un arreglo, se debe acumular lo que se vaya encontrando en el arreglo de forma progresiva. Para esto, es necesario visitar cada posición del arreglo con un ciclo, generalmente se usa el ciclo PARA.

En un arreglo con indexación en base 1, la presente sería la forma general del algoritmo:

```
acum_valores=0
```

```
Para posic=1 hasta tam_arreglo
    acum_valores=acum_valores+arreglo[posic]
```

```
FinPara
```

```
promedio=acum_valores/tam_arreglo
```

Si el arreglo estuviera indexado en base 0, la 1era posición a visitar sería la 0 y la última sería tam_arreglo-1.

El siguiente ejemplo muestra un módulo cuyo objetivo es calcular el promedio de notas de una sección de N alumnos, que están almacenadas en arreglo "notas", donde están almacenadas las notas finales obtenidas por c/u de los alumnos que conforman la sección:

```
Subproceso Calc_PromNotas(N,notas,prom_not Por Referencia)
    Definir posic como entero // variable local
    Definir acum_notas como real // variable local

    acum_notas =0
    Para posic=1 hasta N
        |   acum_notas=acum_notas+notas[posic]
    FinPara
    prom_not=acum_notas/N
FinSubProceso
```

15.2.- Determinando Mayores y Menores Usando Arreglos

Los algoritmos para determinar el elemento mayor y el elemento menor que se vieron en el capítulo 9, están diseñados para ir comparando el dato (respecto al cual se desea determinar el elemento mayor/menor) que se acaba de obtener (por lectura o algún otro tipo de determinación, por ej: un cálculo) contra la variable auxiliar hasta que finalice el ciclo. Esto implica, que no se preservan los valores tomados por el dato en cada iteración, a lo más al final del ciclo sólo se tendrá accesible el último valor tomado por el dato.

En cambio, usando arreglos, los valores tomados por el dato a ser comparado con la variable auxiliar, al estar almacenados en el arreglo, pueden evaluarse en cualquier momento que se deseé o requiera.

Como en la mayoría de algoritmos de arreglos, éste debe recorrerse usando 1 ciclo, bien sea para determinar el elemento mayor o el menor, o ambos, lo cual pudiera hacerse dentro de un módulo que dentro del ciclo se hagan las 2 comparaciones de rigor y se actualice las variables auxiliares correspondientes.

15.3.- Determinar Mayores

En el siguiente ejemplo se tienen cargados en un arreglo las notas de N alumnos y se necesita determinar cuál es la mayor nota. En la función, se asume que el mayor valor se encuentra en la 1era posición del arreglo, por lo cual el "Para" empieza a recorrer a partir de la 2da posición y dentro del ciclo se hace la comparación pertinente. Cuando se cumpla la condición, se actualiza la variable auxiliar mayor_nota. Si existen varios valores iguales al mayor (empate), no importa, porque solo se desea saber cual es el mayor valor, no cuantas veces está presente.

```

Funcion mayor_nota <- Func_Ident_Mayor(N,notas)
    Definir posic como entero

        mayor_nota=notas[1]
        Para posic=2 hasta N
            Si notas[posic] > mayor_nota
                mayor_nota=notas[posic]
            FinSi
        FinPara
    FinFuncion

```

Habiendo determinado el mayor valor de un arreglo, también es muy común, informar en cual(es) posición(es) se encontró el elemento mayor. Continuando con el ejemplo anterior, sabiendo cual es la mayor_nota (para lo cual se invoca a la función), puede recorrerse el arreglo para listar las posiciones donde se encuentra las coincidencias con ese valor e indicar cuantos elementos son iguales a ese elemento mayor.

```

mayor_nota=Func_Ident_Mayor(N,notas)
cantEstMayNota=0
Mostrar "Posiciones donde se encuentra " Sin Saltar
Mostrar "la mayor nota (" mayor_nota " ptos)"
Para posic=1 hasta N
    Si notas[posic]=mayor_nota
        Mostrar "Posición " posic " de " N
        cantEstMayNota=cantEstMayNota+1
    FinSi
FinPara
Mostrar cantEstMayNota " alumnos obtuvieron la mayor nota"

```

15.4.- Determinar Menores

Dándole continuidad al uso del arreglo contentivo de las notas de los N alumnos, se ejemplifica a través de la presente función, la determinación del menor valor, asumiendo también que en la posición 1 está la menor_nota. En esta función, la existencia de varios valores iguales al menor (empate), no importa, porque solo se desea saber cual es el menor valor, no cuantas veces está presente.

```
Funcion menor_nota <- Func_Ident_Menor(N,notas)
    Definir posic como entero

        menor_nota=notas[1]
        Para posic=2 hasta N
            Si notas[posic] < menor_nota
                menor_nota=notas[posic]
            FinSi
        FinPara
    FinFuncion
```

Una vez se conoce el valor del menor valor, gracias a la invocación a la función que se acaba de explicar, se puede mostrar en cual(es) posición(es) se encontró y en cuantas oportunidades. Dándole continuidad al ejemplo con el arreglo de notas, la porción del algoritmo que satisface los objetivos es la siguiente:

```
menor_nota=Func_Ident_Menor(N,notas)
cantEstMenNota=0
Mostrar "Posiciones donde se encuentra " Sin Saltar
Mostrar "la menor nota (" menor_nota " ptos)"
menor_nota=notas[1]
Para posic=1 hasta N
    Si notas[posic]=menor_nota
        Mostrar "Posición " posic " de " N
        cantEstMenNota=cantEstMenNota+1
    FinSi
FinPara
Mostrar cantEstMenNota " alumnos obtuvieron la menor nota"
```