CADP Examen de Promocion

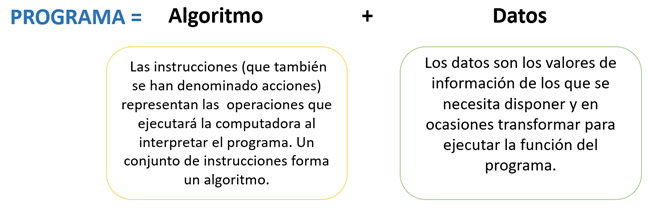
Paradigmas de programación: formas o estilos de programar el cual los lenguajes asumen (Documentado no personal del programador).

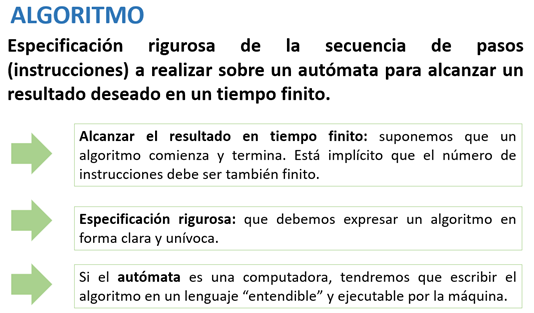
Los lenguajes de programación pueden ser clasificados a partir del modelo que siguen para DEFINIR y OPERAR información. Este aspecto permite jerarquizarlos según el paradigma que siguen.

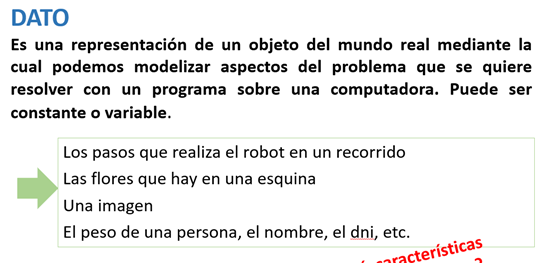
1. Funcional
2. Imperativo
3. Lógico
4. Orientado a objetos

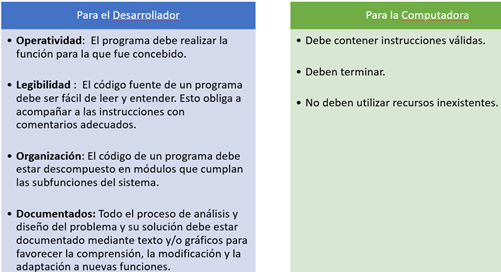
Forma de trabajar:

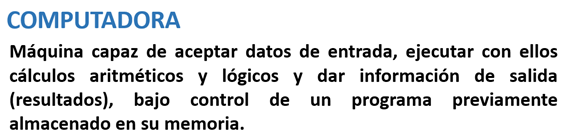
1. **Poseer un problema**.
2. **Modelizar el problema**: El modelo define los mecanismos de interacción y sus condiciones. Establece el efecto sobre la maquina y el usuario. Indica los informes necesarios. Pensar que acciones se van a permitir y que implica cada acción permitida.
3. **Modularizar la solución**: A partir del modelo encontrar la forma de descomponer en partes (módulos) para obtener una solución. El descomponer en módulos reduce la complejidad, distribuye el trabajo y nos sirve por si en el futuro queremos reutilizar código.
4. **Realizar el programa:** Una vez que se tiene la descomposición en funciones/procesos o módulos, debemos diseñar su implementación: esto requiere escribir el programa y elegir los datos a representar.









1. **Utilizar la computadora**: 

Tipos de datos: Clase de objetos de datos ligados a un conjunto de operaciones para crearlos y manipularlos.

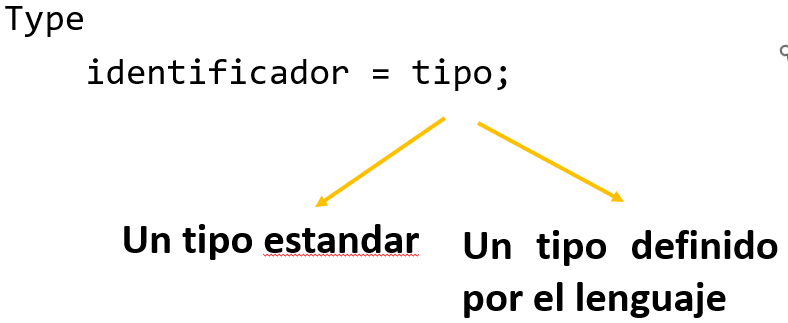
1. Tienen un rango de valores posibles.
2. Tienen una representación interna.
3. Tienen un conjunto de operaciones permitidas.

Clasificación:

1. **Simples**: Toman 1 UNICO valor, en un momento determinado, de todos los permitidos para ese tipo.
2. **Compuesto**: Toman VARIOS valores a la vez que guardan alguna relación lógica entre ellos, bajo un único nombre.

**Definidos por el Lenguaje**: Provistos por el lenguaje y tanto la representación como sus operaciones y valores son reservadas al mismo.

**Definidos por el Programador**: Permiten definir nuevos tipos de datos a partir de los tipos simples. No existe en la definición del lenguaje, y el programador es el encargado de su especificación.



**Beneficios:**

Aumento de la riqueza expresiva del lenguaje, con mejores posibilidades de abstracción de datos.

Mayor seguridad respecto de las operaciones que se realizan sobre cada clase de datos.

Límites preestablecidos sobre los valores posibles que pueden tomar las variables que corresponden al tipo de dato.

**Flexibilidad**: en el caso de ser necesario modificar la forma en que se representa el dato, sólo se debe modificar una declaración en lugar de un conjunto de declaraciones de variables.

**Documentación**: se pueden usar como identificador de los tipos, nombres auto explicativos, facilitando de esta manera el entendimiento y lectura del programa.

**Seguridad**: se reducen los errores por uso de operaciones inadecuadas del dato a manejar, y se pueden obtener programas más confiables.

Los diferentes tipos de datos deben especificarse y a esta especificación dentro de un programa se la conoce como declaración.

Una vez declarado un tipo podemos asociar al mismo variables, es decir nombres simbólicos que pueden tomar los valores característicos del tipo.

Algunos lenguajes exigen que se especifique a qué tipo pertenece cada una de las variables. Verifican que el tipo de los datos asignados a esa variable se correspondan con su definición. Esta clase de lenguajes se denomina fuertemente tipados (strongly typed).

Otra clase de lenguajes, que verifica el tipo de las variables según su nombre, se denomina auto tipados (self typed).

Existe una tercera clase de lenguajes que permiten que una variable tome valores de distinto tipo durante la ejecución de un programa. Esta se denomina dinámicamente tipados (dinamically typed).

**Dato Numérico**: Representa el conjunto de números que se pueden necesitar. Estos números pueden ser enteros o reales.

1. **Entero**: Es un tipo de dato simple, ordinal (puedo saber el valor que le antecede y el que le precede) con una representación interna la cual hace que tenga un máximo y un mínimo. Ej: -10. 200, -3000, 2560.

Operaciones permitidas:

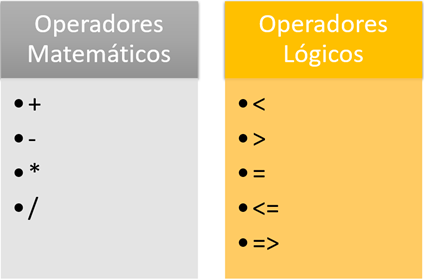


MOD: da el resto entero de la división.

DIV: da el cociente entero de la división

1. **Real**: Es un tipo de dato simple, que permite representar números con decimales y al tener una representación interna, tiene un máximo y un mínimo. Ej: -22.43, 11.5, 200.0, 560.

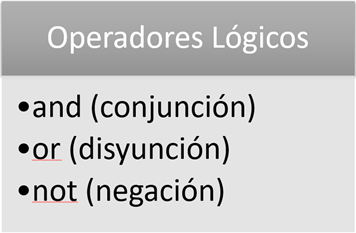
Operaciones permitidas:



**Dato Lógico**: Permite representar datos que pueden tomar dos valores verdadero o falso.

1. **Booleano**: Es un tipo de dato simple, ordinal. Tiene 2 valores posibles: True o False.

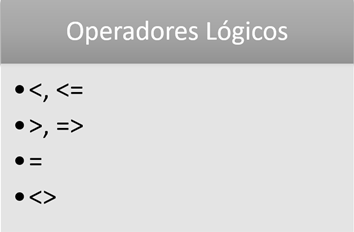
Operaciones permitidas:



**Dato Carácter**: Representa un conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato de tipo carácter contiene solo un carácter.

1. **Char**: Es un tipo de dato simple, ordinal. Ej: a B ¡ $ L 4.

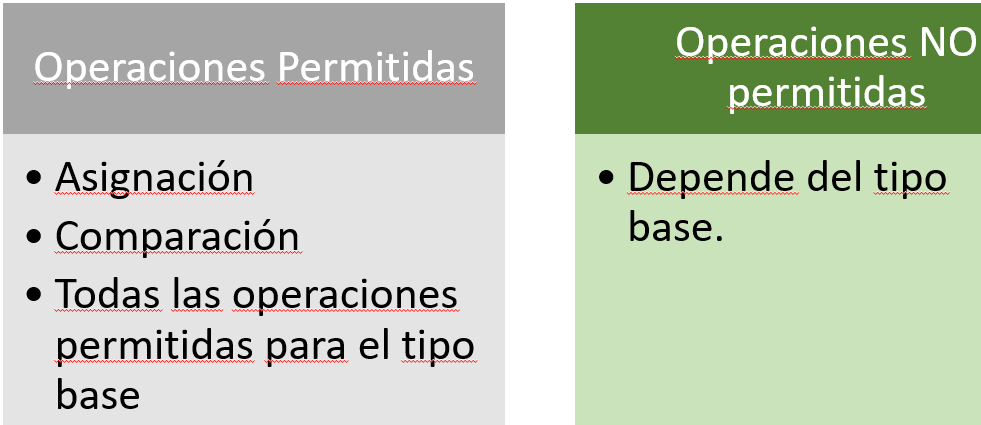
Operaciones permitidas:



El orden esta dado por la tabla ASCII

**Subrango:** Es un tipo simple, ordinal que consiste de una sucesión de valores de un tipo ordinal (predefinido o definido por el usuario) tomado como base.

**Operaciones permitidas:**



**String:** Tipo de dato compuesto definido por el lenguaje. Es una cadena de caracteres. Ej: ‘Juan’ ‘1234’ ‘#$’

**Puntero:** Es un tipo de variable usada para almacenar una dirección en memoria. En esa dirección de memoria se encuentra el valor que puede ser de cualquiera de los tipos vistos (char, boolean, integer, real, string, registro, arreglo u otro puntero).

Caracteristicas:

Es un tipo de dato simple, definido por el lenguaje que contiene la dirección donde se encuentra almacenado el dato real.

Pueden apuntar solamente a direcciones almacenadas en memoria dinámica (heap).

Cada variable de tipo puntero puede apuntar a un único tipo de dato.

Una variable de tipo puntero se indica con ^ y ocupa 4 bytes de memoria (stack) para su representación interna en Pascal. No importa el tipo de dato al que apunta siempre ocupa lo mismo.

Una variable de tipo puntero puede reservar y liberar memoria durante la ejecución de un programa.

**Creación:** Reservar una dirección memoria dinámica libre para poder asignarle contenidos a la dirección que contiene la variable de tipo puntero. new(variable tipo puntero).

**Eliminación:** Liberar la memoria dinámica que contenía la variable de tipo puntero. dispose(variable tipo puntero).

Libera la conexión que existe entre la variable y la posición de memoria. Libera la posición de memoria. La memoria liberada puede utilizarse en otro momento del programa.

**Liberación:** Cortar el enlace que existe con la memoria dinámica. La misma queda ocupada pero ya no se puede acceder.

Nil

Libera la conexión que existe entre la variable y la posición de memoria. La memoria sigue ocupada. La memoria no se puede referenciar ni utilizar.

**Asignación:** Asignar la dirección de un puntero a otra variable puntero del mismo tipo. :=

**Contenido:** Implica poder acceder al contenido que contiene la dirección de memoria que tiene una variable de tipo puntero. ^

Recordar

if (p = nil) then, compara si el puntero p no tiene dirección asignada.

if (p = q) then, compara si los punteros p y q apuntan a la misma dirección de memoria.

if (p^ = q^) then, compara si los punteros p y q tienen el mismo contenido.

**No** se puede hacer **read (p)** ni **write(p)**, si p es una variable de tipo puntero.

No se puede asignar una dirección a un puntero de manera manual (p:= ABCD).

**No** se puede comparar por mayor o menor direcciones de punteros **(p>q)**

Estructura de Datos:

Permite al programador definir un tipo al que se asocian diferentes datos que tienen valores lógicamente relacionados y asociados bajo un nombre único.

**Clasificación de Estructuras de Datos:**



1. **Elementos:** Depende si los elementos son del mismo tipo o no.
   1. **Homogénea:** Los elementos que la componen son del mismo tipo
   2. **Heterogénea:** Los elementos que la componen pueden ser de distintos tipos.
2. **Tamaño:** Hace referencia a si la estructura puede variar su tamaño durante la ejecución del programa.
   1. **Estática:** El tamaño de la estructura no varía durante la ejecución del programa.
   2. **Dinámica:** El tamaño de la estructura puede variar durante la ejecución del programa.
3. **Acceso:** Hace referencia a como se pueden acceder a los elementos que la componen.
   1. **Secuencial:** Para acceder a un elemento particular se debe respetar un orden predeterminado, por ejemplo, pasando por todos los elementos que le preceden, por ese orden.
   2. **Directo:** Se puede acceder a un elemento particular, directamente, sin necesidad de pasar por los anteriores a él, por ejemplo, referenciando una posición.
4. **Linealidad:** Hace referencia a como se encuentran almacenados los elementos que la componen.
   1. **Lineal**: Está formada por ninguno, uno o varios elementos que guardan una relación de adyacencia ordenada donde a cada elemento le sigue uno y le precede uno, solamente.
   2. **No Lineal:** Para un elemento dado pueden existir 0, 1 ó más elementos que le suceden y 0, 1 ó más elementos que le preceden.

**Tipos de Estructuras de Datos:**

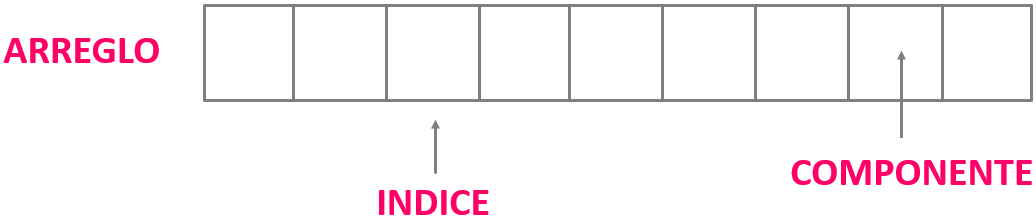
1. **Registro:** Es uno de los tipos de datos estructurados, que permiten agrupar diferentes clases de datos en una estructura única bajo un sólo nombre. Compuesto, Definido por el programador.
   1. **Heterogénea**: Los elementos pueden ser de distinto tipo (puede haber registros con todos elementos del mismo tipo)
   2. **Estática**: El tamaño no cambia durante la ejecución (se calcula en el momento de compilación)
   3. **Campos:** Representan cada uno de los datos que forman el registro.

**Operaciones permitidas:**

**Asignación:** La única operación permitida es la asignación entre dos variables del mismo tipo.

La única forma de acceder a los campos es variable.nombrecampo (Puedo realizar las operaciones permitidas según el tipo de campo del registro)

1. **Arreglos:** Un arreglo (ARRAY) es una estructura de datos compuesta, definida por el programador que permite acceder a cada componente por una variable índice, que da la posición de la componente dentro de la estructura de datos.



**Vector:** Es una colección de elementos que se guardan consecutivamente en la memoria y se pueden referenciar a través de un índice.

1. **Homogénea:** Los elementos pueden son del mismo tipo.
2. **Estática:** El tamaño no cambia durante la ejecución (se calcula en el momento de compilación).
3. **Indexada:** Para acceder a cada elemento de la estructura se debe utilizar una variable ‘índice’ que es de tipo ordinal.

Los elementos son del mismo tipo. Precisamente por ser estática, permite el acceso rápido a sus componentes a través de la variable índice (que tiene que ser de tipo ordinal) y que puede verse como el desplazamiento desde la posición inicial de comienzo de la estructura.

**Rango:** De tipo Ordinal: Integer, Char, Boolean, Subrango.

**Tipo:** Integer, Real, Char, Boolean, Subrango, Registro, Vector.

**Recorridos:** Consiste en recorrer el vector de manera total o parcial, para realizar algún proceso sobre sus elementos.

**Recorrido – Total:** Implica analizar todos los elementos del vector, lo que lleva a recorrer completamente la estructura.

**Recorrido – Parcial:** Implica analizar los elementos del vector, hasta encontrar aquel que cumple con lo pedido. Puede ocurrir que se recorra todo el vector.

**Dimensiones:**

**Dimensión Física:** Se especifica en el momento de la declaración y determina su ocupación máxima de memoria. La cantidad de memoria total reservada no variará durante la ejecución del programa.

**Dimensión Lógica:** Se determina cuando se cargan contenidos a los elementos del arreglo. Indica la cantidad de posiciones de memoria ocupadas con contenido real. Nunca puede superar la dimensión física.

**Agregar:**

1- Verificar si hay espacio (cantidad de elementos actuales es menor a la cantidad de elementos posibles)

2- Agregar al final de los elementos ya existentes el elemento nuevo.

3- Incrementar la cantidad de elementos actuales.

**Insertar:**

1- Verificar si hay espacio (cantidad de elementos actuales es menor a la cantidad de elementos posibles)

2- Verificar que la posición sea válida (esté entre los valores de dimensión definida del vector y la dimensión lógica).

3- Hacer lugar para poder insertar el elemento.

4- Incrementar la cantidad de elementos actuales.

**Eliminar:**

1- Verificar que la posición sea válida (esté entre los valores de dimensión definida del vector y la dimensión lógica).

2- Hacer el corrimiento a partir de la posición y hasta el final.

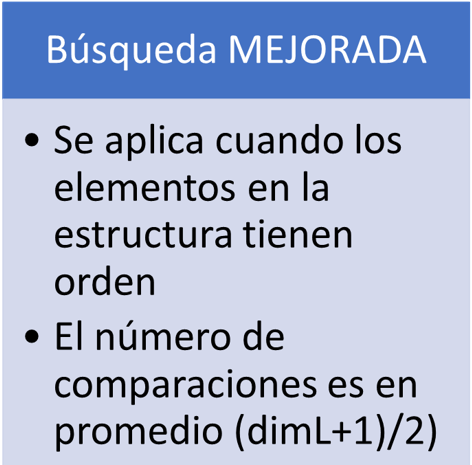
3- Decrementar la cantidad de elementos actuales.

**Buscar:**

**Vector Desordenado:** Se debe recorrer todo el vector (en el peor de los casos), y detener la búsqueda en el momento que se encuentra el dato buscado o que se terminó el vector.

**Vector Ordenado:** Se debe aprovechar el orden, existen al menos dos formas:

**Búsqueda mejorada:**



**Búsqueda dicotómica:**

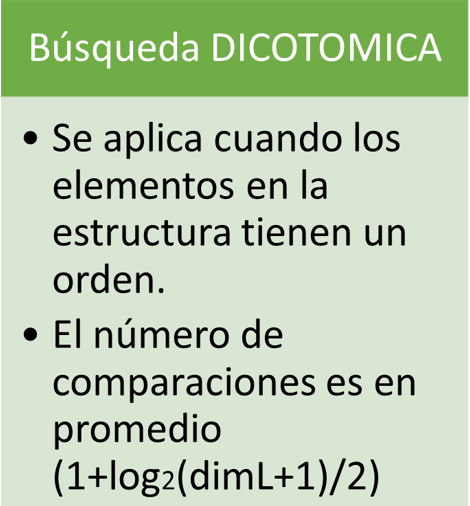
Se calcula el elemento que esta en la posición del medio

Si es el elemento que busco, entonces la búsqueda termino

Si NO es el elemento que busco, entonces

Comparo contra el valor del medio

Elijo del vector la mitad que me convenga



1. **Lista:** Colección de nodos, donde cada nodo contiene un elemento y en qué dirección de memoria se encuentra el siguiente nodo. Cada nodo de la lista se representa con un puntero, que apunta a un dato (elemento de la lista) y a una dirección (donde se ubica el siguiente elemento de la lista). Toda lista tiene un nodo inicial.

Los nodos que la componen pueden no ocupar posiciones contiguas de memoria. Es decir, pueden aparecer dispersos en la memoria, pero mantienen un orden lógico interno.

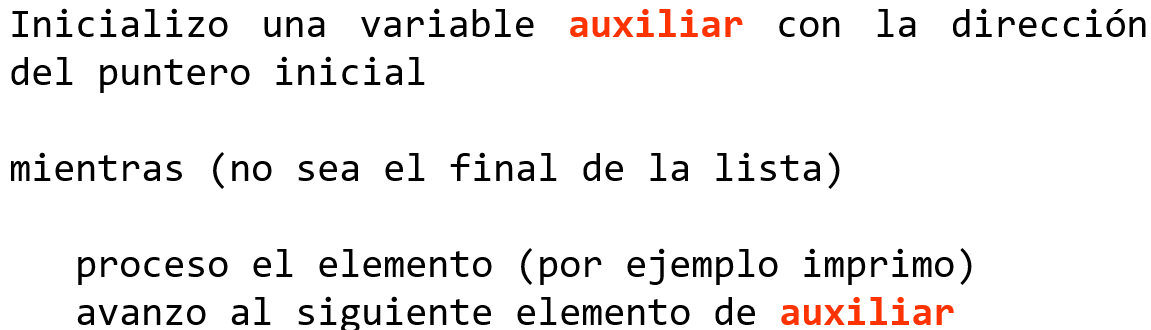
**Características**:

1. **Homogénea**: Los elementos son todos del mismo tipo.
2. **Dinámica**: La cantidad de nodos puede variar durante la ejecución
3. **Lineal**: Cada nodo tiene un único antecesor y sucesor.
4. **Acceso:** El acceso a cada elemento es de manera secuencial

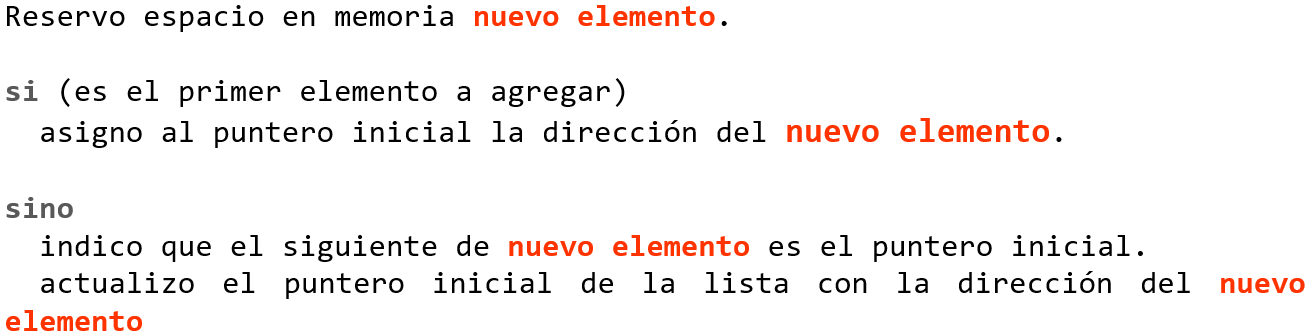
Cada vez que se necesite agregar un nodo se deberá reservar memoria dinámica (new) y cuando se quiera eliminar un nodo se debe liberar la memoria dinámica (dispose).

**Crear una lista**: Implica marcar que la lista no tiene una dirección inicial de comienzo. **Lista: = nil.**

**Recorrer una lista:** Implica posicionarse al comienzo de la lista y a partir de allí ir “pasando” por cada elemento de la misma hasta llegar al final.



**Agregar adelante en una lista:** Implica generar un nuevo nodo y agregarlo como primer elemento de la lista.

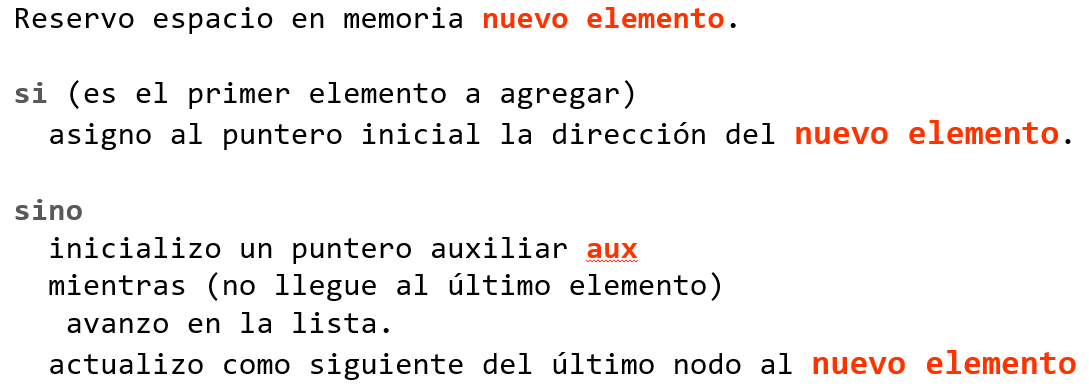


**Agregar al final en una lista:** Implica generar un nuevo nodo y agregarlo como último elemento de la lista.

Genero espacio para el nuevo elemento

Recorro la lista hasta llegar al último elemento.

Reasigno los punteros



Problema: Cada vez que debo agregar un nuevo elemento al final de la lista, se tiene que recorrer la misma de manera completa hasta llegar al final.

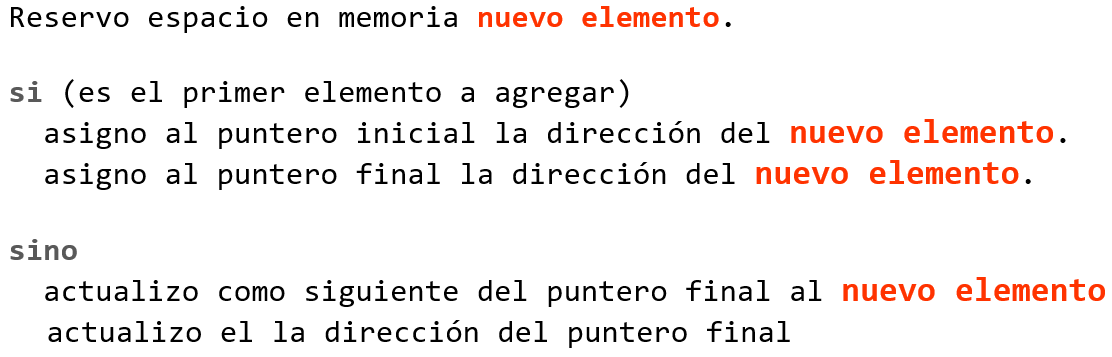
**Agregar al final en una lista opción 2:**

Genero espacio para el nuevo elemento

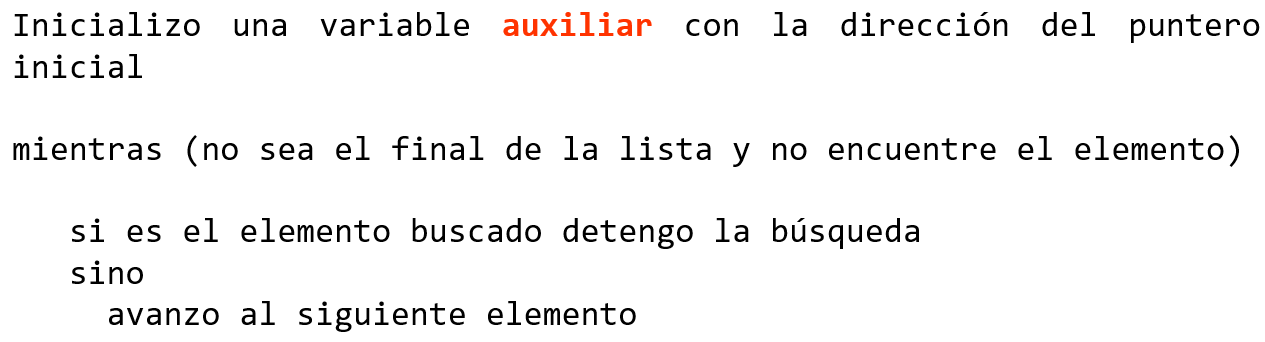
Utilizo un puntero que mantiene la dirección del último elemento.

Reasigno los punteros.

Reasigno la dirección del último elemento



**Buscar un elemento**: Implica recorrer la lista desde el comienzo pasando nodo a nodo hasta encontrar el elemento buscado o que se termine la lista.



**Insertar un elemento:** Se necesita que la estructura tenga un orden e implica agregar el elemento a la lista de manera que la misma siga ordenada.

**Existen 4 casos:**

1.La lista está vacía. Si la lista NO está vacía: se debe recorrer hasta encontrar el lugar donde va el elemento:

2.El elemento a insertar va al comienzo de la lista.

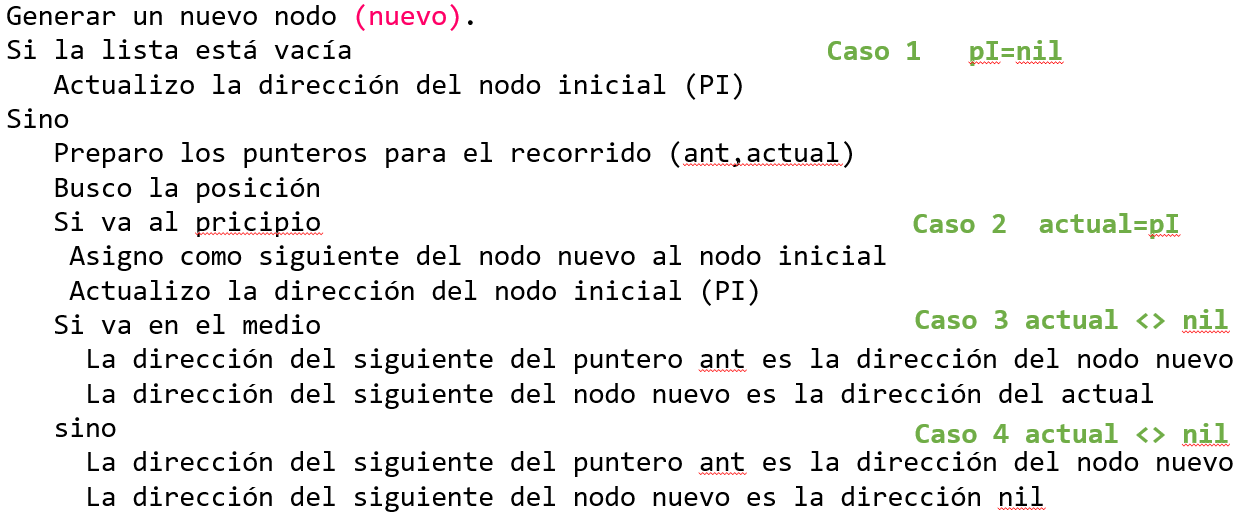
3.El elemento a insertar va al medio de la lista.

4.El elemento a insertar va al final de la lista.

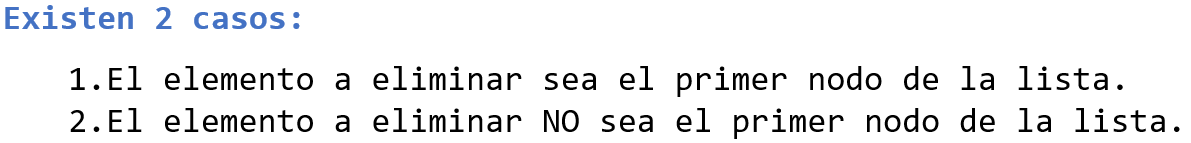
**Caso 1:** La lista esta vacia: Generar un nuevo nodo (nuevo). Asignar a la dirección del puntero inicial (PI) la del nuevo nodo (nuevo).

**Caso 2**: la lista no está vacía, y el elemento va al principio de la lista: Generar un nuevo nodo (nuevo). Asignar a la dirección del puntero siguiente del nuevo la dirección del nodo inicial (PI). Actualizar con la dirección del nuevo nodo la dirección del puntero inicial (PI).

**Caso 3**: la lista no está vacía, y el elemento va en el medio de la lista. Generar un nuevo nodo (nuevo). Preparo los punteros para el recorrido. Busco la posición. Reasigno punteros.



**Eliminar un elemento**: Implica recorrer la lista desde el comienzo pasando nodo a nodo hasta encontrar el elemento y en ese momento eliminarlo (dispose). El elemento puede no estar en la lista.



**Caso 1:** el elemento está al comienzo de la lista.

Actualizar la dirección siguiente del puntero inicial pI con la dirección siguiente de actual. Actualizar la dirección del puntero inicial pI. Hacer el dispose.

**Caso 2:** El elemento NO esta al comienzo de la lista.

Actualizar la dirección siguiente del puntero ant con la dirección siguiente de actual. Hacer el dispose.

# Tipos de Variables:

1. **Variables**: Zona de memoria cuyo contenido va a ser alguno de los tipos mencionados anteriormente. La dirección inicial de esta zona se asocia con el nombre de la variable. Puede cambiar su valor durante el programa.
   1. **Alcance de las variables**:
      1. **Variables Globales**: Pueden ser usadas en todo el programa (incluyendo módulos)
      2. **Variables locales al programa**: Pueden ser usadas sólo en el cuerpo del programa

Se busca si es variable local al programa

Se busca si es variable global al programa

* + 1. **Variables locales al proceso**: Pueden ser usadas sólo en el proceso que están declaradas

Se busca si es variable local

Se busca si es un parámetro

Se busca si es variable global al programa

1. **Constante**: Zona de memoria cuyo contenido va a ser alguno de los tipos mencionados anteriormente. La dirección inicial de esta zona se asocia con el nombre de la variable. **NO** puede cambiar su valor durante el programa.

**Precondición**: Es la información que se conoce como verdadera antes de iniciar el programa (ó módulo).

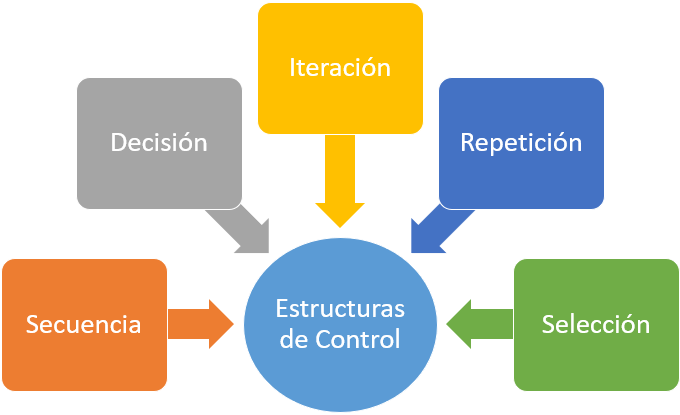
**Postcondición**: Es la información que debería ser verdadera al concluir el programa (ó módulo), si se cumplen adecuadamente los pasos especificados.

# Operaciones:

1. Read: Se usa para tomar datos desde un dispositivo de entrada (por defecto desde teclado) y asignarlos a las variables correspondientes.
2. Write: Se usa para mostrar el contenido de una variable, por defecto en pantalla.

# Estructuras de Control:

Todos los lenguajes de programación tienen un conjunto mínimo de instrucciones que permiten especificar el control del algoritmo que se quiere implementar. Como mínimo deben contener: secuencia, decisión e iteración.



1. **Secuencia:** La estructura de control más simple, está representada por una sucesión de operaciones (por ej. asignaciones), en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones.
   1. **Repetición**: Es una extensión natural de la secuencia. Consiste en repetir N veces un bloque de acciones. Este número de veces que se deben ejecutar las acciones es fijo y conocido de antemano. La estructura FOR es Repetitiva.

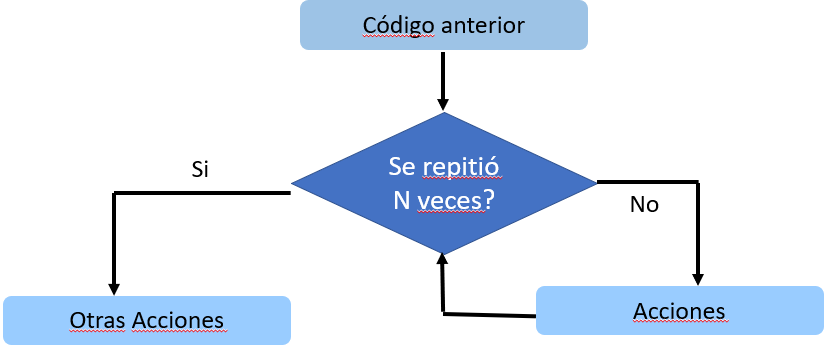
**Importante:**

La variable índice debe ser de tipo ordinal.

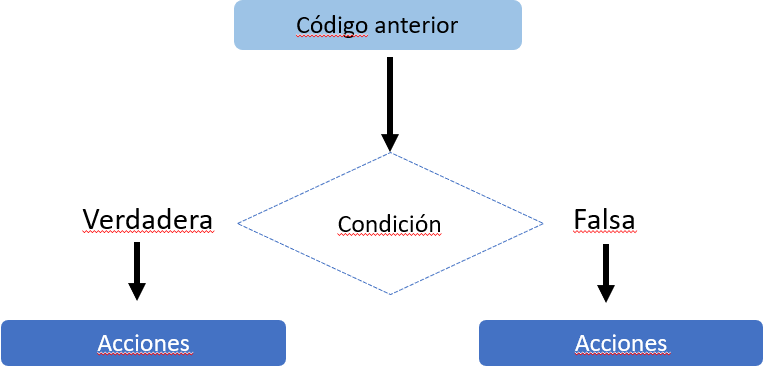
La variable índice no puede modificarse dentro del lazo

La variable índice se incrementa y decrementa automáticamente

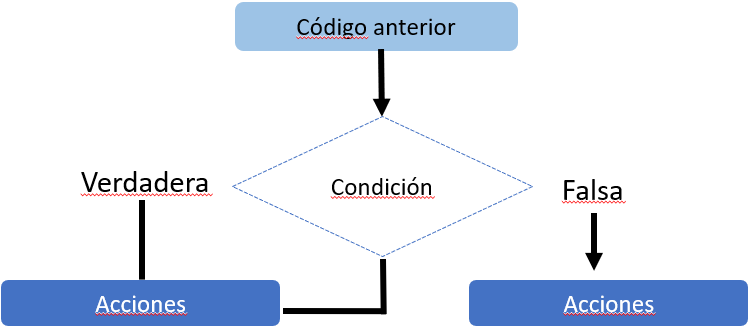
Cuando el for termina la variable índice no tiene valor definido



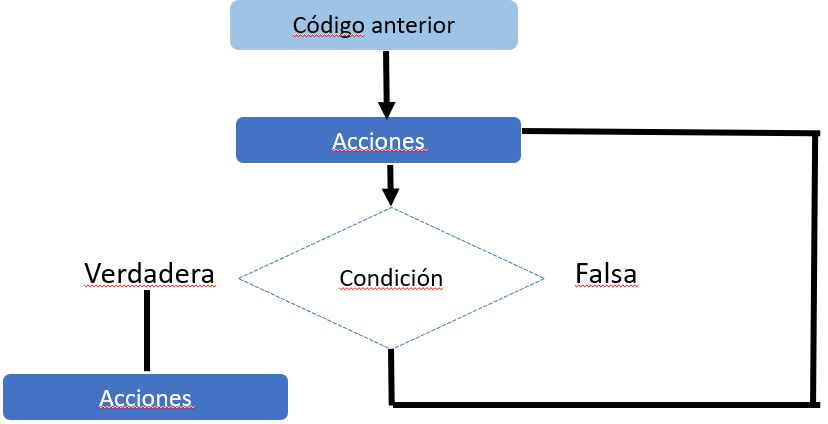
1. **Decisión**: En un algoritmo representativo de un problema real es necesario tomar decisiones en función de los datos del problema. La estructura de control IF es una de decisión. La estructura básica de decisión entre dos alternativas es la que se representa simbólicamente:



1. **Iteración:** Puede ocurrir que se desee ejecutar un bloque de instrucciones desconociendo el número exacto de veces que se ejecutan. Para estos casos existen en la mayoría de los lenguajes de programación estructurada las estructuras de control iterativas condicionales. Como su nombre lo indica las acciones se ejecutan dependiendo de la evaluación de la condición. Estas estructuras se clasifican en **pre-condicionales** y **post-condicionales**.
   1. **Precondicional:** Evalúan la condición y si es verdadera se ejecuta el bloque de acciones. Dicho bloque se pueda ejecutar 0, 1 ó más veces. Importante: el valor inicial de la condición debe ser conocido o evaluable antes de la evaluación de la condición. La estructura WHILE es precondicional.



* 1. **Postcondicional**: Ejecutan las acciones luego evalúan la condición y ejecutan las acciones mientras la condición es falsa. Dicho bloque se pueda ejecutar 1 ó más veces. La estructura REPEAT-UNTIL es Postcondicional.

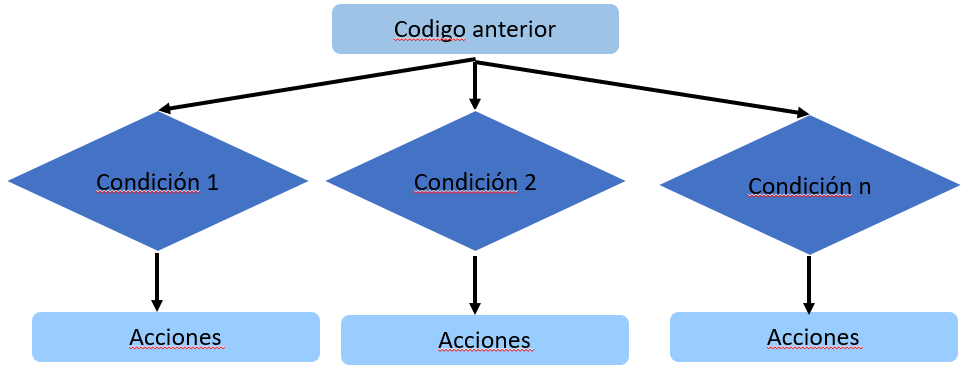


1. **Selección:** Permite realizar distintas acciones dependiendo del valor de una variable de tipo ordinal. La estructura CASE es de tipo Selección.

**Importante:**

La variable del case debe ser de tipo ordinal

Las opciones deben ser disjuntas



# Modularizar:

Significa dividir un problema en partes funcionalmente independientes, que encapsulen operaciones y datos.

No se trata simplemente de subdividir el código de un sistema de software en bloques con un número de instrucciones dado.

Separar en funciones lógicas con datos propios y datos de comunicación perfectamente especificados.

Cada subproblema está en un mismo nivel de detalle y puede resolverse independientemente.

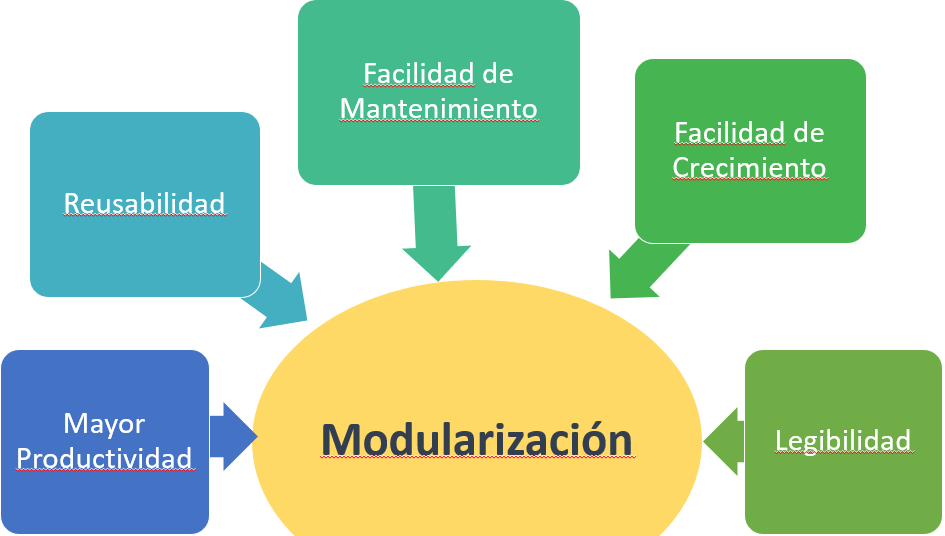
Las soluciones de los subproblemas pueden combinarse para resolver el problema original.

**Modulo**: Tarea específica bien definida se comunican entre sí adecuadamente y cooperan para conseguir un objetivo común.

Encapsula acciones tareas o funciones.

En ellos se pueden representar los objetivos relevantes del problema a resolver.

Existen diferentes metodologías para usarlos en los programas en particular nosotros usaremos la **METODOLOGIA TOP-DOWN.**



**Mayor Productividad**: reduciendo el tiempo de desarrollo global del sistema

**Reusabilidad:** Posibilidad de utilizar repetidamente el producto de software desarrollado. Naturalmente la descomposición funcional que ofrece la modularización favorece el reúso.

**Facilidad de Crecimiento**: Disminuir los riesgos y costos de incorporar nuevas prestaciones a un sistema en funcionamiento.

**Legibilidad**: Mayor claridad para leer y comprender el código fuente.

## Estructuras para modularizar en pascal:

**Procedure**: Conjunto de instrucciones que realizan una tarea especifica y retorna 0, 1 ó más valores.

**Function**: Conjunto de instrucciones que realizan una tarea especifica y retorna un **único valor** de tipo **SIMPLE**.

Invocación:

Usando variable

En un While/If

En un Write

Comunicación entre Módulos:

1. **Variables Globales**: Demasiados identificadores. No se especifica la comunicación entre los módulos. Conflictos de nombres de identificadores utilizados por diferentes programadores. Posibilidad de perder integridad de los datos, al modificar involuntariamente en un módulo datos de alguna variable que luego deberá utilizar otro módulo.
2. **Parámetros**: La solución a estos problemas ocasionados por el uso de variables globales es una combinación de ocultamiento de datos (Data Hiding) y uso de parámetros. El ocultamiento de datos significa que los datos exclusivos de un módulo NO deben ser "visibles" o utilizables por los demás módulos. El uso de parámetros significa que los datos compartidos se deben especificar como parámetros que se trasmiten entre módulos.

**Comunicación:**

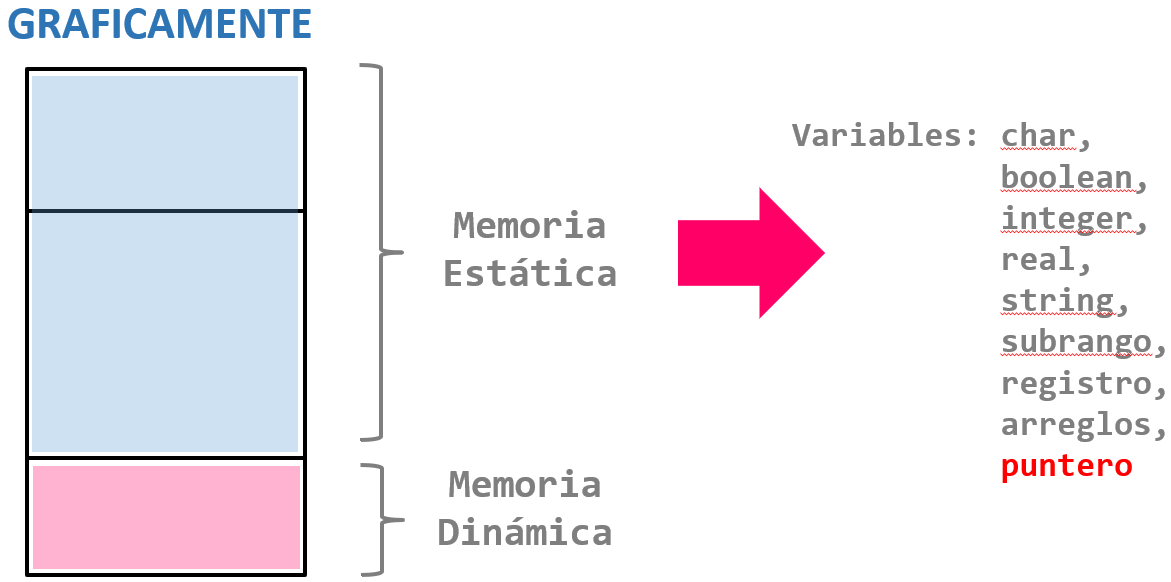
El número y tipo de los argumentos utilizados en la invocación a un módulo deben coincidir con el número y tipo de parámetros del encabezamiento del módulo.

Un parámetro por valor debiera ser tratado como una variable de la cual el módulo hace una copia y la utiliza localmente. Algunos lenguajes permiten la modificación local de un parámetro por valor, pero toda modificación realizada queda en el módulo en el cual el parámetro es utilizado.

El número y tipo de los argumentos utilizados en la invocación a un módulo deben coincidir con el número y tipo de parámetros del encabezamiento del módulo

* 1. Se analiza para cada módulo entonces: ¿cuáles son los datos propios? y ¿cuáles son los datos compartidos?
  2. Los datos propios se declararán locales al módulo.
  3. Los datos compartidos se declararán como parámetros (Por Valor o Por Referencia):
     1. **Parámetro por Valor**: Un dato de entrada por valor es llamado parámetro IN y significa que el modulo recibe (sobre una variable local) un valor proveniente de otro modulo (o del programa principal). Con él puede realizar operaciones y/o cálculos, pero no producirá ningún cambio ni tampoco tendrá incidencia fuera del módulo. HACE UNA COPIA DEL VALOR ENVIADO.
     2. **Parámetro por Referencia**: La comunicación por referencia (OUT, INOUT) significa que el módulo recibe el nombre de una variable (referencia a una dirección) conocida en otros módulos del sistema. Puede operar con ella y su valor original dentro del módulo, y las modificaciones que se produzcan se reflejan en los demás módulos que conocen la variable. COMPARTE LA DIRECCIÓN DE MEMORIA.

# Memoria Estática y Dinámica:



**Variables Estáticas:** No permiten modificar su tamaño en tiempo de ejecución. Las variables y tipos reservan memoria en su declaración y se mantienen durante todo el programa. El lenguaje puede validar previo a la ejecución

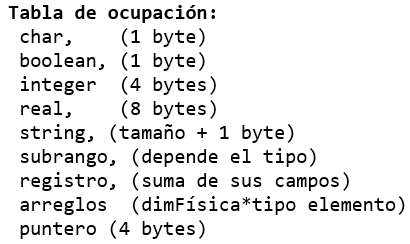
**Variables Dinámicas:** Permiten modificar en tiempo de ejecución la memoria utilizada.

## Cálculo de memoria:

Memoria necesaria para la ejecución de un programa puede dividirse en dos.

**MEMORIA ESTATICA**: a modo de simplicidad consideraremos sólo las **variables locales** y **variables globales** de programa.

**MEMORIA DINAMICA:** a modo de simplicidad consideraremos sólo cuando en la ejecución de un programa **se reserva** o **libera memoria**. new(), dispose() y nil.



# Máximos y Mínimos:

Explicación de como un programa debería calcular el máximo y mínimo:

1. Utilizar una variable que representará al máximo/mínimo.
2. Inicializar la variable antes de comenzar la lectura de los datos. El máximo en un valor bajo. El mínimo en un valor alto.
3. Actualizar la variable máximo/mínimo cuando corresponda

# Corrección y Eficiencia:

Cuando se desarrollan los algoritmos hay dos conceptos importantes que se deben tener en cuenta:

**Corrección:** Un programa es correcto si se realiza de acuerdo a sus especificaciones.

**Técnicas para corrección de programas:**

* 1. **Testing:** El propósito del Testing es proveer evidencias convincentes que el programa hace el trabajo esperado.

**Diseñar un plan de pruebas:**

Decidir cuales aspectos del programa deben ser testeados y encontrar datos de prueba para cada uno de esos aspectos.

Determinar el resultado que se espera que el programa produzca para cada caso de prueba.

Poner atención en los casos límite.

Diseñar casos de prueba sobre la base de lo que hace el programa y no de lo que se escribió del programa. Lo mejor es hacerlo antes de escribir el programa.

**Una vez que el programa ha sido implementado y se tiene el plan de pruebas:**

Se analizan los casos de prueba.

Si hay errores se corrigen.

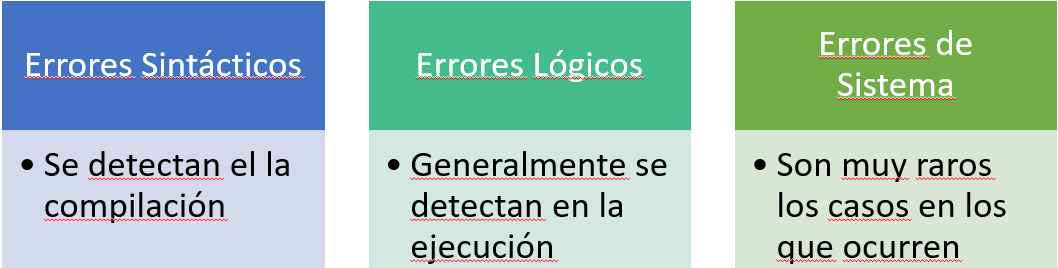
Estos dos pasos se repiten hasta que no haya errores.

* 1. **Debugging:** Es el proceso de descubrir y reparar la causa del error. El diseño y aplicación de pruebas adicionales para ubicar y conocer la naturaleza del error. Es agregar sentencias adicionales en el programa para poder monitorear su comportamiento más cercanamente.

**Los errores pueden provenir de dos fuentes:**

El diseño del programa no es el adecuado.

El programa no está escrito correctamente.

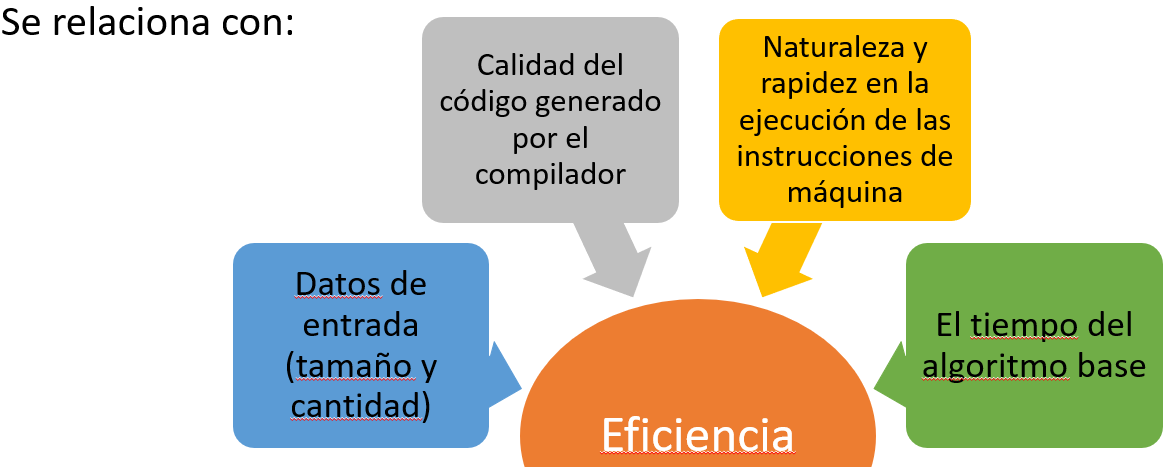


* 1. Walkthroughs: Es recorrer un programa frente a una audiencia. La lectura de un programa a alguna otra persona provee un buen medio para detectar errores. Esta persona no comparte preconceptos y está predispuesta a descubrir errores u omisiones. A menudo, cuando no se puede detectar un error, el programador trata de probar que no existe, pero mientras lo hace, puede detectar el error, o bien puede que el otro lo encuentre.
  2. Verificación: Verificar un programa significa controlar que se cumplan las pre y post condiciones del mismo.

PARA DETERMINAR LA CORRECCION DE UN PROGRAMA PUEDO UTILIZAR UNO, DOS, TRES O LAS CUATRO TECNICAS DE CORRECCION

Una vez que se obtiene un algoritmo y se decide que es correcto, es importante determinar la eficiencia del mismo.

**Eficiencia:** El análisis de la eficiencia de un algoritmo estudia el tiempo que tarda un algoritmo en ejecutarse y la memoria que requiere.



**Se divide en:**

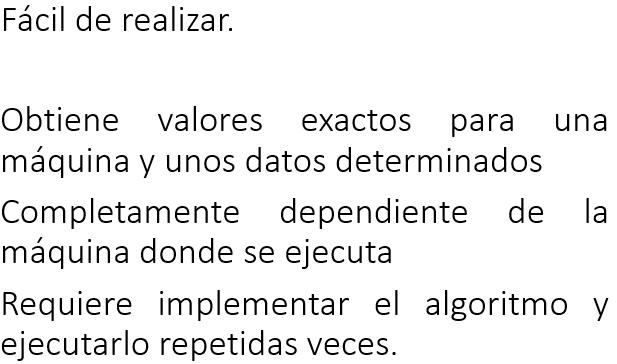
**Tiempo de ejecución:** El tiempo de un algoritmo puede definirse como una función de entrada:

Existen algoritmos que el tiempo de ejecución tiempo de ejecución no depende de las características de los datos de entrada sino de la cantidad de datos de entrada o su tamaño

Existen otros algoritmos el tiempo de ejecución es una función de la entrada “específica”, en estos casos se habla del tiempo de ejecución del “peor” caso. En estos casos, se obtiene una cota superior del tiempo de ejecución para cualquier entrada.

**Puede calcularse haciendo dos tipos de análisis:**

**Análisis empírico:** Para realizar un análisis empírico, es necesario realizar el programa y medir el tiempo consumido.



**Análisis teórico:** Implica encontrar una cota máxima para expresar el tiempo de nuestro algoritmo, sin necesidad de ejecutarlo.

Dado un algoritmo que es correcto se calcula el tiempo de ejecución de cada una de sus instrucciones. Para eso se va a considerar:

Sólo las instrucciones elementales del algoritmo: asignación, y operaciones aritmético/lógicas.

Una instrucción elemental utiliza un tiempo constante para su ejecución, independientemente del tipo de dato con el que trabaje. 1UT.

Se considera que cada operación elemental se ejecuta en una unidad de tiempo.

**Asignación:** 1 Unidad de Tiempo

**Operaciones Aritméticas:** 1 Unidad de Tiempo.

**If:** 1 Unidad de Tiempo por Condición + cuerpo.

**If-else:** 1 Unidad de Tiempo por Condición + Max(cuerpo-if, cuerpo-else)

**For:** 3.(N) + 2 + N.(cuerpo). N número máximo del for.

While: N + 1. (evaluar condición) + N.(cuerpo)