

Conceptos de Algoritmos, Datos y Programas

¿Qué es la **Informática**?

Ciencia que estudia la resolución de problemas del mundo real mediante el uso de una computadora.

Etapas de resolución de un problema

1. Obtener un problema : Investigar el problema.
2. Obtener un modelo del problema : Modelizar el problema
3. Modularizar : Descomponer el problema en partes.
4. Realizar programa : Diseñar implementación de módulos y elegir datos a representar.
5. Utilizar computadora : Ejecutar el programa para obtener resultados.

Algoritmo: conjunto de pasos / instrucciones que se ejecuta en una computadora para obtener un resultado en un tiempo finito.

Estas instrucciones le indican a la computadora cómo realizar tareas específicas, como cálculos, procesamiento de datos o control de dispositivos externos.

Datos: es la información que se procesa y manipula por un programa.

Clase de objeto que tiene su conjunto de operaciones, tiene un rango de valores definidos, y una representación interna.

¿Qué partes componen un programa?

Programa: Algoritmos + Datos

Un programa en informática es una secuencia de instrucciones escritas en un lenguaje de programación que una computadora puede interpretar y ejecutar.

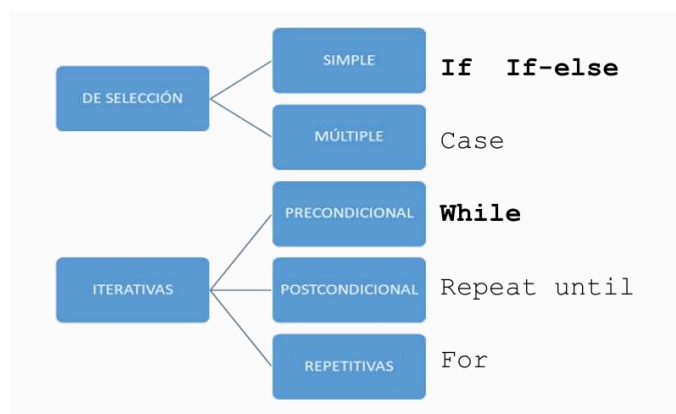
Precondición: Información que se conoce como verdadera antes de iniciar el programa

Post-condición: Información verdadera que se conoce al finalizar un programa

Estructura de control: Componente fundamental en la programación que permite controlar el flujo de ejecución, como el orden en que se ejecuta y las condiciones que debe cumplir.

Tipos de estructuras de control:

1. **Secuencia:** sucesión de operaciones que se ejecutan según el orden físico en el que aparecen las instrucciones.
2. **Iteración:** Se ejecuta 0,1 o más veces. Depende de la evaluación de una condición. pre o post condicional.
3. **Decisión:**
4. **Selección :** Evalúa decisiones en función de los datos del problema.
5. **Repetitivas:** se repiten una cantidad conocida de veces.



IF / IF - ELSE: (SI / SINO) si cumple condición simple entra a la instrucción.

CASE: si cumple condición múltiple entra a la instrucción.

WHILE: Mientras cumpla esta condición ejecuta la instrucción.

REPEAT UNTIL: Hasta que no se cumpla la condición ejecuta el bucle.

FOR: Ejecuta el bucle n cantidad de veces.

Las estructuras de control en programación se pueden clasificar en varias categorías basadas en su comportamiento y cómo controlan el flujo de ejecución del programa.

¿Qué es un dato?

Un dato es la representación de una variable o constante, que puede ser tanto cualitativa como cuantitativa. Además, requiere una interpretación para convertirse en información.

Constante: No varía durante la ejecución del programa.

Variable: Varía durante la ejecución del programa.

DIF entre variable global y local

Variable global: se puede usar durante todo el programa incluyendo módulos. Abarca el programa de principio a fin.

Variable local: solo se puede utilizar dentro del módulo donde fue declarada.

Cuales de los siguientes conceptos se ven **afectados** con el uso de **variables globales**

- Protección de datos: se alteran los valores a medida que se ejecuta el programa.

- Cantidad de memoria utilizada: el espacio utilizado es menor que declarar varias variables locales. Por otra parte, el espacio de la variable local se libera al finalizar el módulo.
- Tiempo de ejecución del algoritmo: no se ve afectado.
- Corrección de programa: Puede efectuarse, con variables globales se corre el riesgo de alterar valores y que no funcione.
- Cantidad de líneas de código del programa: no hay cambio notable.
- Mantenimiento posterior: es más fácil mantener un programa con variables locales, ya que con variable global se puede alterar el funcionamiento de los módulos.

Ocultamiento y protección de datos:

Ocultamiento: Los datos son exclusivos de un módulo. No debe ser visible o utilizable por los demás módulos.

Protección de datos: se refiere a que otros módulos no alteren un valor no deseado.

Alcance de los datos:

El alcance es una propiedad de las variables. Se refiere a la región del programa donde se puede utilizar la variable.

Las variables a su vez tienen distintos alcances.

¿Cómo vinculas los tres conceptos anteriores?

Los datos dentro de un módulo se pueden proteger mediante el ocultamiento y este a su vez es determinado por el alcance, es decir, el alcance de la variable sería solo dentro del módulo.

Si se permite que los datos sean accedidos desde cualquier módulo puede dificultar el trabajo en equipo, ya que se debería determinar antes el nombre de cada variable.

Tipos de Datos

La forma en la que se presenta la información, puede variar y están ligados a diferentes conjuntos de operaciones para crearlos y manipularlos. Como, rango de valores posibles, operaciones permitidas y representación interna.

Diferencia entre **datos estándar de un lenguaje** y **tipos definidos por el usuario**

Dato estándar de un lenguaje: es un conjunto de valores los cuales ya viene predefinido por el lenguaje, es decir, las operaciones permitidas y la representación interna.

Dato definido por el usuario: Un tipo de dato no estándar es aquel que no existe en la definición del lenguaje y el programador se encarga de su especificación mediante el uso de dato simple.

Ventajas:

- Flexibilidad a la hora de querer representar el dato
- Facilita la documentación ya que puede nombrarse de una manera que auto explique su uso
- Es más seguro ya que se reducen los errores por el uso de operaciones inadecuadas al manejar el dato

COMPUESTO: pueden tomar varios valores a la vez que guardan alguna relación lógica entre ellos, bajo un único nombre.

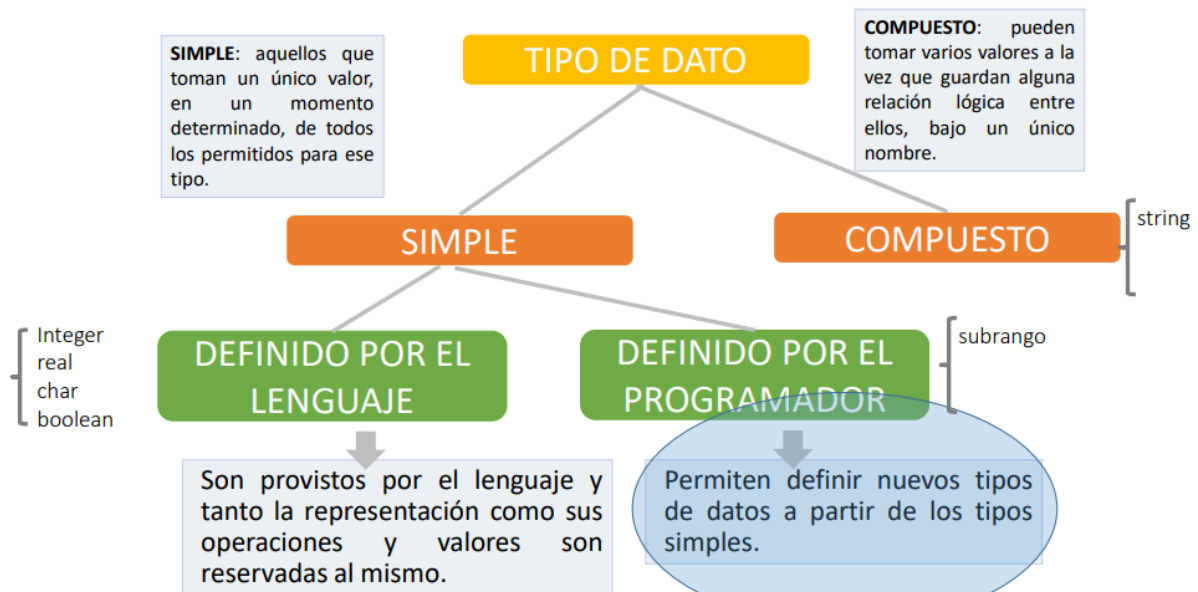
SIMPLE: aquellos que toman un único valor, en un momento determinado, de todos los permitidos para ese tipo.

- **DEFINIDO POR EL LENGUAJE**

Son provistos por el lenguaje y tanto la representación como sus operaciones y valores son reservadas al mismo.

- **DEFINIDO POR EL PROGRAMADOR**

Permiten definir nuevos tipos de datos a partir de los tipos simples.



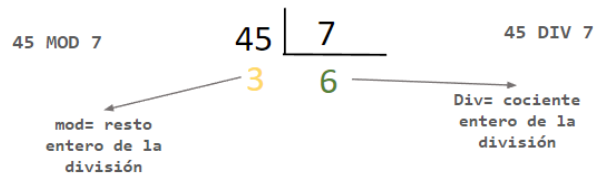
Dato numérico

- tipo de datos **entero (integer)**

Es un tipo de dato simple, ordinal (se sabe quien lo antecede y su siguiente)

Operadores Matemáticos	Operadores Lógicos	Operadores Enteros
<ul style="list-style-type: none"> • + • - • * • / 	<ul style="list-style-type: none"> • < • > • = • <= • => 	<ul style="list-style-type: none"> • Mod • Div

DIV vs MOD



- tipo de datos **real (real)**

Es un tipo de dato simple (no se aplica div y mod)

Operadores Matemáticos	Operadores Lógicos
<ul style="list-style-type: none">• +• -• *• /	<ul style="list-style-type: none">• <• >• =• <=• ==>

Orden de precedencia para resolución

1. Operadores: * , / , div, mod
2. Operadores: + , -

Recuerde que para imprimir en pantalla números reales puede utilizar la notación:

writeln(X:Y:Z) donde X es el número a imprimir, Y es el ancho (en cantidad de caracteres) que debe ocupar la impresión, y Z es la cantidad de decimales. Por ejemplo, sea el número

pi=3.141592654 :

3.14 => writeln(pi,1,2);

3.14 => writeln(pi,8,2); (hay 4 espacios delante del 3, para completar 8 caracteres de ancho)

3,1415 => writeln(pi,1,4);

Dato lógico

tipo de dato simple, ordinal.

- Los valores son de la forma **(boolean)**:

true = verdadero

false = falso

Operadores Lógicos

- and (conjunción)
- or (disyunción)
- not (negación)

V	V	V
F	F	F
V	F	F
F	V	F

Conjunción

V	V	V
F	F	F
V	F	V
F	V	V

Disyunción

V	F
F	V

Negación

Dato Carácter

- Tipo de datos lógico (**char**): tipo de dato simple y ordinal (ASCII)

Los valores son de la forma: a B ! \$ L 4

Operadores Lógicos

- <, <=
- >, >=
- =
- <>

Dato String

Conjunto de caracteres, al igual que char se pueden realizar operaciones lógicas. Es un tipo de dato compuesto.



Continúa más adelante..(Tipos de datos def por usuario)

Tipos de Variables:

Variables: puede cambiar su valor mientras el programa se va ejecutando.

Constante: NO puede cambiar su valor durante el programa.

Estructura de un programa hasta ahora:

```
Program nombre;      {nombre}
Const
    N = 25;          {Constantes del programa}
    pi = 3.14;
módulos
    ...              {Módulos del programa}
var
    edad: integer;
    peso: real;      {variables}
    letra: char;
    resultado: boolean;
begin
    edad:= 5;
    peso:= -63.5;    {cuerpo del programa}
    edad:= edad + N;
    letra:= 'A';
    resultado:= letra = 'a';
end.
```

¿Cómo se da valor a una variable?

- usando :=
- mediante operación lectura : **read (variable)/ write (variable)**

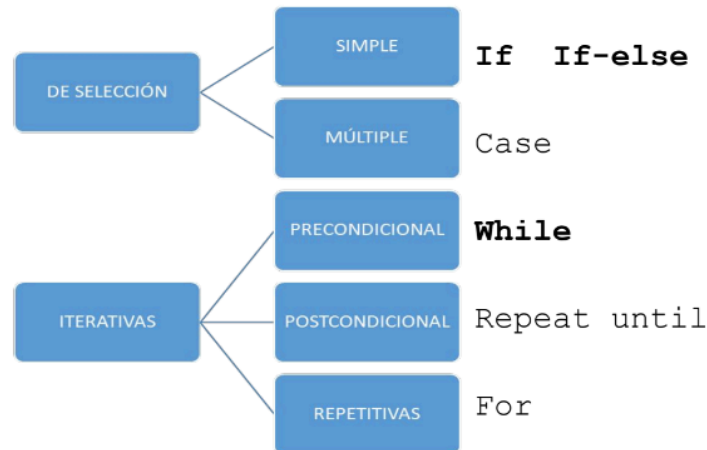
Read (leer): Se usa para tomar datos desde un dispositivo de entrada y asignarlos a las variables.

readln: el cursor queda debajo de la frase.

Write (escribir): Se usa para mostrar el contenido de una variable en pantalla (informar) el cursor queda al lado de la frase.

writeln: el cursor queda debajo de la frase.

Estructuras de control (con ejemplos)



1. **Secuencia**: sucesión de operaciones en la que el orden de ejecución coincide con el orden físico.

Program uno;

...

var

num:integer;

begin

read (num);

write (num);

end.

2. **Decisión**: algoritmo en donde es necesario que tome 2 decisiones en función de los datos del problema.

if (condición) **then**

acción 1

else

acción 2;

3. **Iteración**:

- Pre Condicional(while): mientras la condición es verdadera se ejecuta el bloque de acciones. Se repite mientras la condición sea verdadera. Puede ejecutarse 0, 1 o más veces.

while (condición) **do**

accion;

- Post condicional: ejecuta una vez y después evalúa la condición. Se repite mientras la condición es falsa. Puede ejecutarse 1 o más veces.

```
repeat
    accion;
until (condición)
```

4. **Repetición**: Se ejecuta una cantidad de veces conocida de antemano.

```
for indice := valor_inicial to valor_final do
    accion 1;
```

(cuando el límite primero es más grande que el segundo)

```
for indice := valor_final downto valor_inicial do
    accion 1;
```

(en caso de que se recorra de un número mayor a un número menor)
ej:

```
For indice := 20 downto 18 do
begin
    accion1;
    accion2;
end.
```

Problema: se leen valores de alturas de personas, hasta leer la altura 1.59. Informar la cantidad de personas que miden entre 1.00 y 1.30; la cantidad de personas que miden entre 1.31 y 1.50; la cantidad de personas que miden entre 1.51 y 1.89 y las que miden más de 1.89

El alumno 1: utiliza una variable real para leer las alturas y cuatro contadores para contar la cantidad de personas en cada rango.

Además utiliza un while como estructura de control principal y dentro utiliza un case que incluye los rangos de alturas para saber cual contador sumar. Al final informa los valores de los contadores.

```
program Alumno1
var
    alt:real;
```

```

    r1,r2,r3,r4:integer;
begin
    read(alt)
    r1:=0; r2:=0; r3:=0; r4:=0;
    while (alt < '1,59') do begin
        case alt of
            '1.00'..'1.30': r1:= r1+1;
            '1.31'..'1.50': r2:= r2+1;
            '1.51'..'1.89': r3:= r3+1;
            '1.89' : r4:= r4+1;
            else
                end;
        end;
    end;
    write ()
End.

```

El alumno 2: utiliza una variable real para leer las alturas y cuatro contadores para contar la cantidad de personas en cada rango.

Además utiliza un while como estructura de control principal y dentro utiliza un if con else para saber cual contador sumar. Al final informa los valores de los contadores.

```

program alumno2
var
    alt: real;
    r1, r2, r3, r4: integer;
begin
    read(alt);
    r1:=0; r2:=0; r3:=0; r4:=0;
    while (alt<'1.59') do begin
        if (alt= '1.00').. or (alt='1.30') then
            r1:=r1 + 1
        else
            if (alt='1.31')..or(alt='1.50') then
                r2:=r2 + 1
            else
                if....
            else

```

```
    read(alt);
end;
write( );
End.
```

Máximos

```
Program uno;
var
  prom:real;alu:integer; max:real;maxalu:integer;
begin
  read(prom);
  read(alu);
  max:= -1;
  while (prom <> 0) do
    begin
      If (prom >= max) then begin
        max:= prom;
        maxalu:= alu;
      end;
      read(prom);
      read(alu);
    end;
  write ("El mejor alumno es:", maxalu );
end.
```

Mínimos

```
Program uno;
var
  prom:real;alu:integer; min:real;
begin
  read(prom);
  read(alu);
  min:=11;
  while (prom <> 0) do
    begin
      If (prom <= min) then
        min:= prom;
      read(prom);
      read(alu);
    end;
  write ("El mejor promedio es:", min );
end.
```

Tipos de datos definidos por el usuario

Subrango

Operaciones Permitidas

- Asignación
- Comparación
- Todas las operaciones permitidas para el tipo base

```
Program uno;  
Type  
  letras = 'a'..'z';  
var  
  primer,segundo,tercer:integer;  
  letra:letras;  
Begin  
  primer:= 0;  segunda:=0;  tercer:=0;  
  read (letra);  
  while (letra <> 'z') do  
    begin  
      case letra of  
        'a'..'h': primer:= primer + 1;  
        'i'..'n': segunda:= segunda + 1;  
        'ñ'..'y': tercer:= tercer + 1;  
      end;  
      read (letra);  
    end;  
  write (primer,segunda,tercer);  
end.
```

Type

identificador=tipo;

...

Var

x:identificador;

Modularización

La programación estructurada trata de dividir el programa en bloques más chicos que funcionen independientemente.

La idea es que los módulos trabajen en conjunto para resolver el problema por partes.

- VENTAJAS: mayor productividad, usabilidad, facilidad de mantenimiento/ crecimiento, legibilidad.

¿Cómo diseñarías una solución modularizada adecuadamente?

Separando el programa en funciones lógicas con datos propios y que a su vez los módulos envíen los datos necesarios para cumplir el objetivo.

¿Qué consideraciones tendrías en cuenta al descomponer en módulos?

- Cada subproblema en el mismo nivel.
- El subproblema puede resolverse independientemente.
- Combinar soluciones de subproblemas para resolver el problema general.

REUSABILIDAD: Los módulos reutilizables están diseñados de manera que puedan ser integrados y utilizados en diversos contextos, lo que ahorra tiempo y esfuerzo en el desarrollo de software al evitar la necesidad de escribir código repetitivo.

¿Cómo relacionarías la modularización con la reusabilidad y el mantenimiento de los sistemas de software?

Favorece el reuso de software desarrollado.

La modularización promueve la reusabilidad al crear componentes autónomos con interfaces claras, y ambos conceptos juntos facilitan el mantenimiento al aislar los cambios y reducir la complejidad del sistema. Esto resulta en sistemas de software más robustos, flexibles y fáciles de gestionar a lo largo del tiempo.

La división lógica de un sistema en módulos permite aislar los errores que se producen con mayor facilidad, esto significa corregir los errores en menos tiempo y disminuye los costos de mantenimiento del sistema.

PROCEDIMIENTOS

Conjunto de instrucciones que realizan una tarea específica y retorna 0, 1 ó más valores. ejecuta una serie de acciones que están relacionadas entre sí, y no devuelve ningún valor o múltiples valores, y escribir en pantalla y leer datos

Procedimientos sencillos

```
procedure Acceso;
```

Procedimientos con argumentos

```
procedure saludo (nombre: string);
```

Procedimientos con parámetros

```
procedure modifica( variable : integer);
```

```
procedure modifica( var variable : integer);
```

```
procedure modifica( dato : integer);
```

Program uno;

Const

....

Type

....

```
procedure auxiliar;
```

```
Var
```

```
  x:integer;
```

```
begin
```

```
  x:=8;
```

```
end;
```

```
Var
```

....

```
Begin
```

```
  auxiliar;
```

```
End.
```

¿Es obligatorio **type** en Pascal?

No, la sección **type** no es obligatoria. Solo se usa si necesitas definir **tipos de datos personalizados** como registros (**record**), arreglos (**array** con nombre), conjuntos (**set**), punteros, etc.

FUNCIONES

Conjunto de instrucciones que realizan una tarea específica y retorna 1 valor de tipo simple.

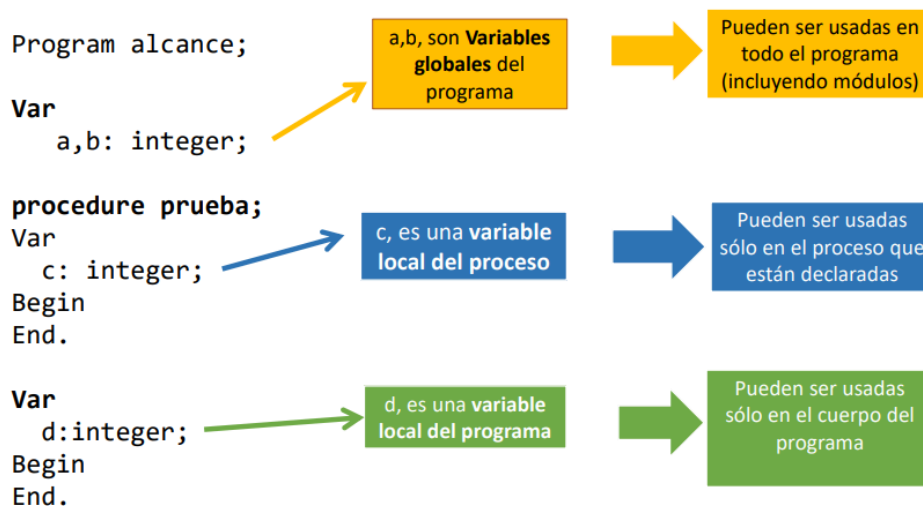
(entero/boolean/char/reales/subrango)

La función devuelve como mínimo 1 valor, y no puede ni escribir en pantalla ni leer datos.

La mayoría de las que vamos a ver son funciones matemáticas que están ya predefinidas en Pascal:

- **Abs**: valor absoluto de un número.
- **Sin**: seno de un cierto ángulo dado en radianes.
- **Cos**: coseno, análogo.
- **ArcTan**: arco tangente. No existe función para la tangente, que podemos calcular como $\sin(x)/\cos(x)$.
- **Round**: redondea un número real al entero más cercano.
- **Trunc**: trunca los decimales de un número real para convertirlo en entero.
- **Int**: igual que trunc, pero el resultado sigue siendo un número real.
- **Sqr**: eleva un número al cuadrado.
- **Sqrt**: halla la raíz cuadrada de un número.
- **Exp**: exponencial en base e, es decir, eleva un número a e.
- **Ln**: calcula el logaritmo neperiano (base e) de un número.
- **Odd**: devuelve TRUE si un número es impar.
- **Potencias**: no hay forma directa de elevar un número cualquiera a otro en pascal, pero podemos imitarlo con "exp" y "ln", así:

Alcance de variables



Si Variable no está inicializada, imprime basura

Si Variable no está declarada, error

si estoy en un proceso

1. se busca si es variable local
2. se busca si es un parámetro
3. se busca si es variable global

si es una variable usada en un ProgramaP

1. se busca si es variable local del programa
2. se busca si es variable local
3. si no es ninguna de las anteriores entonces da error.

Comunicación entre módulos

Parámetros

(Solución al problema de variable globales)

- Valor (entrada): un módulo recibe un valor, lo copia trabaja con el mismo, devuelve y en el valor real no se produce ningún cambio, solo sobre su copia.

Admite tanto procedimientos como funciones

```
procedure uno (nombre1: tipo; nombre2: tipo);
```

```
...
```

```
var
```

```
...
```

```
begin
```

```
    uso parámetros
```

```
End;
```

- Referencia (entrada/salida): recibe una dirección de memoria, realiza operaciones que producirán cambios por fuera del módulo.

Solo utilizamos procedimientos porque no admite funciones al devolver datos.

ej:

```
procedure uno (var nombre1: tipo; var nombre2: tipo);
```

```
...
```

```
var
```

```
...
```

```
begin
```

```
    uso parámetros
```

```
End;
```

Tipos de datos estructurados

Aquellos que permiten agrupar varios elementos bajo un mismo nombre, facilitando el manejo de colecciones o conjuntos de datos. Estos tipos incluyen arreglos, registros, conjuntos

Tipo de dato compuesto

Son aquellos que combinan otros tipos de datos (primitivos o estructurados) para construir estructuras más complejas. Subrango, punteros, clases y objetos.

Clasificación de estructuras de datos

Por los elementos:

- **Homogénea:** Todos los elementos son del mismo tipo (ejemplo: arreglos).
- **Heterogénea:** Los elementos son de distintos tipos (ejemplo: registros).

Por el acceso:

- **Secuencial:** Se recorren los elementos en orden, uno a uno (ejemplo: bucles en arreglos).
- **Directo:** Se accede directamente a un elemento por su índice o clave (ejemplo: `arreglo[3]`).

Por el tamaño:

- **Estática:** Tamaño fijo, definido en tiempo de compilación (ejemplo: arreglos estáticos).
- **Dinámica:** Tamaño puede cambiar en tiempo de ejecución (ejemplo: listas enlazadas).

Por la linealidad:

- **Lineal:** Los datos siguen un orden, con antecedente y sucesor (ejemplo: arreglos, pilas, colas).
- **No lineal:** Los datos no tienen un orden secuencial (ejemplo: árboles, grafos).

Registros

Es un tipo de datos estructurado, que permite agrupar diferentes clases de datos en una estructura única bajo un sólo nombre

Es una estructura de datos:

Heterogénea, estática y tiene campos, que representan cada uno de los datos que forman el registro

La característica principal es que un registro permite representar la información en una única estructura.

Definición de tipo

```
tipoRegistro = record
  campo1: tipo-campo1;
  campo2: tipo-campo2;
end;
```

Declaración de variable

```
miVariable: tipoRegistro;
```

Acceso a los datos

```
miVariable.campo1
```

Operaciones

- Leer un registro: campo a campo
- Imprimir un registro: campo a campo
- Comparar un registro con otro: campo a campo
- Asignar un registro a otro: usando el operador :=

Ejemplos

```

Procedure LeerRegistro (var s:sitio);
begin
  With s do begin
    readln(nombre);
    if (nombre <> 'fin')then begin
      readln(prov);
      readln(contAct);
      readln(cantVis);
    end;
  end;
end;

```

```

Procedure actualizarMax (cantAct: integer; nombreAct:
string; var max: integer; var nommax: string );
begin
  if (cantAct > max) then begin
    max:= cantAct;
    nommax:= nombreAct;
  end;
end;

```

Registros de registros

```

Program uno;
Type
  fecha = record
    dia: integer;
    mes:integer;
    año:integer;
  end;
  perro = record
    raza: string;
    nombre: string;

```

```

    edad: integer;
    fechaVis: fecha;
end;
procedure leerFecha (var f: fecha); lee fecha
Begin
    read(f.dia);
    read(f.mes);
    read(f.año);
end;
procedure leer (var p: perro); lee cada perro
Begin
    read(p.raza);
    read(p.nombre);
    read(p.edad);
    leerFecha(p.fechaVis); proced leo fecha dentro del
otro registro
end;
function cumpleFecha (f: fecha): boolean;
var
    ok: boolean;
begin
    if ( ((f.dia >=1) and (f.dia <=15)) and ((f.mes = 1) or
(f.mes = 2)) and (f.año = 2023) ) then
        ok:= true
    else
        ok:= false;
        cumpleFecha:= ok;
end;
Var
    ani: perro;
    cant,i: integer;
Begin
    cant:=0;
    for i:= 1 to 10 do begin
        leer (ani);
        if (cumpleFecha (ani.fechaVis) = true) then
            cant:= cant + 1;

```

```
end;  
write (`La cantidad es`, cant);  
End.
```

Corte de Control

(estructura ordenada por algún criterio)

```
begin  
  {Programa Principal}  
  nomMax:= '';  
  max:= -1;  
  leer(ciudad Tur);  
  {Se lee 1er registro}  
  While (ciudad Tur.prov <> 'fin') do begin  
    provActual:= ciudadTur.prov; inicializa la prov actual  
    cantidad:=0;  
    while (ciudad Tur.prov <> 'fin') and (ciudadTur.prov =  
provActual) do begin mientras sea la provincia actual realiza  
operaciones  
      cantidad:= cantidad + ciudadTur.cantVis;  
      leer(ciudadTur); lee próximo  
    {Se lee otro registro}  
    end;  
    actualizarMax(cantidad, provActual, max, nomMax);  
  end;  
  writeln('Prov con mayor cant. de visit. es:', nomMax);  
end.
```

```
procedure leer(var c: tciudad);  
begin  
  With c do begin  
    readln(prov);  
    if (prov <> 'fin') then begin deja de leer datos con  
la cond de fin
```



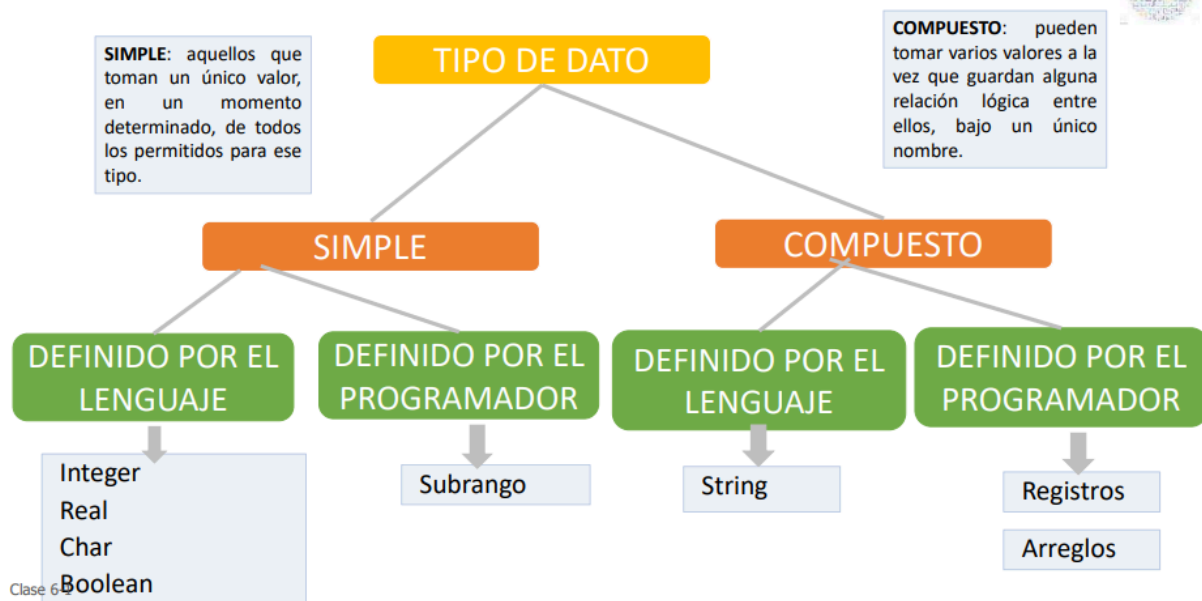
```

    readln(nombre);
    readln(cantAct);
    readln(cantVis);
end;
end;
end;

```

CADP – TIPOS DE DATOS

ESTRUCTURADOS



Arreglos

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

VECTOR (arreglo de una dimensión) Es una colección de elementos que se guardan consecutivamente en la memoria y se pueden referenciar a través de un índice.

- **HOMOGÉNEA**: Los elementos pueden ser del mismo tipo

- ESTÁTICA: El tamaño no cambia durante la ejecución (se calcula en el momento de compilación)
- INDEXADA: para acceder a cada elemento de la estructura se debe utilizar una variable “índice” que es de tipo ordinal (sigue un orden conocido)
- LINEAL : se sabe el elemento anterior y el siguiente de cada elemento.

El **rango** debe ser un tipo ordinal:

char, entero, booleano, subrango.

El **tipo** debe ser un tipo estático:

char, entero, booleano, subrango, real, registro, vector?

```
Program uno;
Const
  ....
Type
  vector = array [rango] of tipo;
Var
  variable: vector;
```

¿Se pueden declarar de la siguiente manera, directamente en el campo de variables locales al módulo?

```
var
  estudiantes: array[1..10000] of TEstudiante;
  irregulares: array[1..1000] of Integer;
  numEstudiantes, numIrregulares, i, j: Integer;
```

En Pascal si es posible declarar arreglos en la sección de variables si se cumplen ciertos requisitos. En este caso solo se podrá trabajar con

la variable en el módulo declarado, no puede pasarse como parámetro.

En esta materia siempre los declaramos en la zona de type.
(consultar a profesor)

Carga de valores

```
Procedure carga (var v: vector);  
var  
    i:integer;  
begin  
    for i:= 1 to tam do  
        read (v[i]);  
    end;
```

```
Procedure imprimir (v: vector);  
var  
    i:integer;  
begin  
    for i:= 1 to tam do  
        write (v[i]);  
    end;
```

Recorridos

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

```
function posición (v: vector): integer;
var
  pos, resto: integer;
  seguir: boolean;
begin
  seguir:= true;
  pos:=1;
  while ((pos<= tam) and (seguir = true)) do begin
    resto:= v[pos] MOD multi;
    if (resto = 0) then
      seguir:= false
    else
      pos:= pos + 1;
    end;
    if (seguir = false) then
      posicion:= pos
    else
      posicion:= -1;
```

```
function verificar (v:vector; valor:integer):integer;
Var
  i,cant:integer;
Begin
  cant:= 0;
  for i:= 1 to tam do begin
    if (valor = v[i]) then
      cant:= cant + 1;
    end;
```

```
    verificar:= cant;  
End;
```

Dimensión física y lógica

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

```
Program uno;  
Const  
    df=10  
Type  
    valores = array [1..df ] of integer;  
Var  
    v: valores;  
    max:integer;  
    dL: integer;  
Begin  
    cargarValores (v , dL );  
    max:= maximo (v , dL);  
End.
```

Procedimiento para cargar array cuando no se llena por completo la dimensión física

```

Procedure cargarValores (var a: números; var
dimL:integer);
Var
    num:integer;
Begin
    dimL:=0;
    read (num);
    while ((dimL < dF) and (num <> 50)) do begin
        dimL:= dimL+1; suma 1 a la dimensión lógica
        a[dimL]:= num; guarda el elemento en array
        read(num); lee prox numero
    end;
End;

```

Agregar elementos al final

- 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.

```

Procedure agregar (var a :números; var dL:integer; var pude:boolean; num:integer);

```

```

Begin
    pude:= false;
    Verifico si hay espacio
    if ((dL + 1) <= física) then
    begin
        pude:= true;
        dL:= dL + 1;
        a[dL]:= num;
    end;
end.

```

Registro que se pudo realizar

Incremento la dimensión lógica

Agrego elelemento

```

Program uno;
const
    fisica = 10;

```

```

    type numeros= array [1..fisica] of integer;
var
    VN: números;
    dimL, valor:integer;
    ok:boolean;
Begin
    cargar (VN,dimL);
    read(valor);
    agregar(VN,dimL,ok,valor);
End.

```

Insertar elementos

Significa agregar en el vector un elemento en una posición determinada. Puede pasar que esta operación no se pueda realizar si el vector está lleno o si la posición no es válida

- 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.

```

var
    VN: numeros;
    dimL, valor, pos: integer;
    ok: boolean;
Begin
    cargar( VN, dimL);
    read (valor);
    read(pos);
    insertar (VN, dimL,ok,num,pos);
End.

```

```

Procedure insertar (var a :números; var dL:integer; var pude:boolean;
Var
    num:integer; pos: integer);
    i:integer;

Begin
    pude:= false;
    if ((dL + 1) <= física) and (pos>= 1) and (pos <= dL) )then begin
        for i:= dL downto pos do
            a[i+1]:= a[i];
        pude:= true;
        a[pos]:= num;
        dL:= dL + 1;
    end;
end;

```

Verifico si hay espacio y si la posición es válida

Corro los elementos empezando desde atrás hasta la posición a insertar para hacer el hueco donde se va a insertar el elemento

Registro que se pudo realizar
Inserto el elemento
Incremento la dimensión lógica

Borrar elementos

Significa borrar (lógicamente) en el vector un elemento en una posición determinada, o un valor determinado. Puede pasar que esta operación no se pueda realizar si la posición no es válida, o en el caso de eliminar un elemento si el mismo no está.

- 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


```

Procedure eliminar (var a :números; var dL:integer; var pude:boolean;pos: integer);
Var
  i:integer;

Begin
  pude:= false;      Verifico si la posición es válida
  if ((pos>= 1) and (pos <= dL) )then begin
    for i:= pos to (dL-1) do
      a[i]:= a[i+1];
    } Corro los elementos empezando desde la posición
      hasta la dimensión lógica-1 para "tapar" el
      elemento a eliminar

    pude:= true;
    dL:= dL - 1;
  } Registro que se pudo realizar
  end;      Decremento la dimensión lógica
end;

```

Búsqueda de un elemento

Significa recorrer el vector buscando un valor que puede o no estar en el vector. Se debe tener en cuenta que no es lo mismo buscar en un vector ordenado que en uno que no lo esté.

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 en el que se terminó el vector.

1- Inicializar la búsqueda desde la posición 1 (pos).

2- Mientras ((el elemento buscado no se igual al valor en el arreglo[pos]) y (no se termine el arreglo))

2.1 Avanzo una posición

3- Determino porque condición se ha terminado el while y devuelvo el resultado.

```

function buscar(a :números; dL:integer;
valor:integer):boolean;
Var
  pos:integer;
  esta:boolean;
Begin

```

```

esta:= false;
pos:=1;
while ( (pos <= dL) and (not esta) ) do begin
    if (a[pos]= valor) then
        esta:= true
    else
        pos:= pos + 1;
    end;
    buscar:= esta;
end.

```

Vector Ordenado

- Se debe recorrer el vector teniendo en cuenta el orden:
 - BÚSQUEDA MEJORADA
- 1- Inicializar la búsqueda desde la posición 1 (pos).
- 2- Mientras ((el elemento buscado sea menor al valor en el arreglo[pos]) y (no se termine el arreglo))
 - 2.1 Avanzo una posición
- 3- Determino porque condición se ha terminado el while y devuelvo el resultado.

```

Function existe (a:números; dL:integer;
valor:integer):boolean;
Var
    pos:integer;
Begin
    pos:=1;
    while ((pos <= dL) and (a[pos]<valor)) do begin
        pos:= pos + 1;
    end;
    if ( (pos <= dL) and (a[pos]= valor)) then
        buscar:=true
    else
        buscar:= false;

```

end.

- BÚSQUEDA BINARIA

1- Se calcula la posición media del vector (teniendo en cuenta la cantidad de elementos)

2- Mientras ((el elemento buscado sea \neq arreglo[medio]) y ($\text{inf} \leq \text{sup}$))

 Si ((el elemento buscado sea $<$ arreglo[medio]) entonces

 Actualizo sup

 Sino

 Actualizo inf

 Calculo nuevamente el medio

3- Determino porque condición se ha terminado el while y devuelvo el resultado.

```
Function dicotomica (a:números; dL:integer; valor:integer):boolean;
```

```
Var
```

```
    pri, ult, medio : integer;
```

```
    ok:boolean
```

```
Begin
```

```
    ok:= false;
```

```
    pri:= 1 ; ult:= dL; medio := (pri + ult ) div 2 ;
```

```
    While ( pri <= ult ) and ( valor  $\neq$  vec[medio]) do
```

```
        begin
```

```
            if ( valor < vec[medio] ) then
```

```
                ult:= medio -1 ;
```

```
            else pri:= medio+1 ;
```

```
            medio := ( pri + ult ) div 2 ;
```

```
        end;
```

```
        if (pri <=ult) and (valor = vec[medio]) then ok:=true;
```

```
    end;
```

```
    dicotomica:= ok;
```

```
end.
```

Punteros

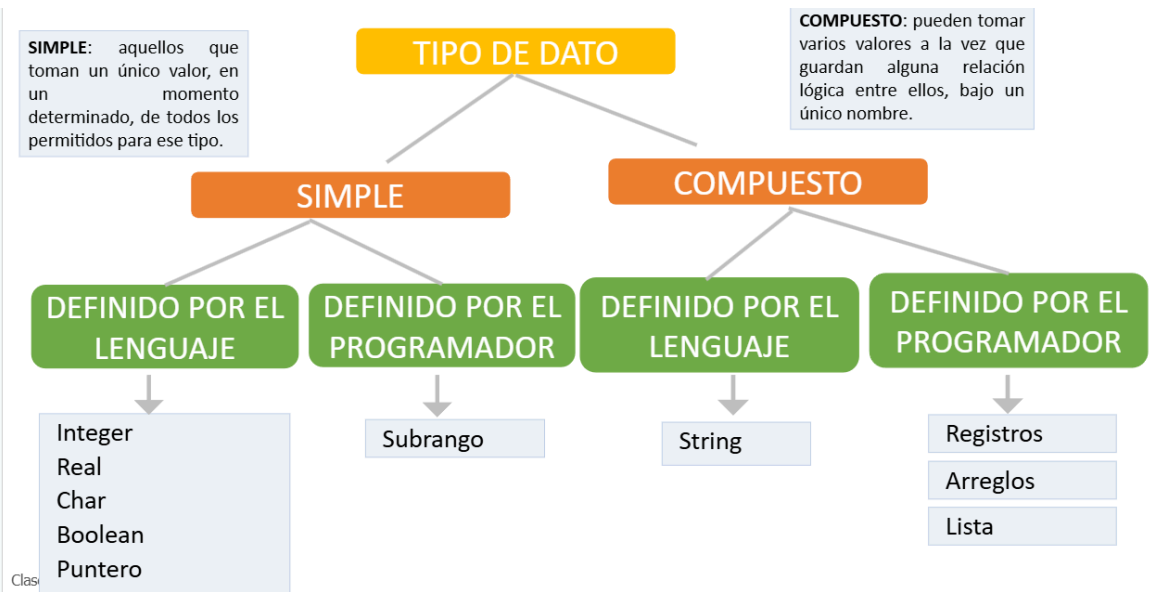
Un **puntero** en Pascal es un tipo de dato que almacena la dirección de memoria de otra variable. En lugar de contener un valor directamente, un puntero contiene la referencia a la ubicación de ese valor en la memoria.

La dirección del puntero se almacena en memoria estática y el elemento al que apunta se encuentra en memoria dinámica.

```
type PEntero = ^Integer; // Puntero a un entero.  
var  
    numero: PEntero;  
begin  
    New(numero); // Reserva memoria para un entero.  
    numero^ := 42; // Asigna el valor 42 a esa memoria.  
    writeln(numero^); // Muestra el valor almacenado(42).  
    Dispose(numero); // Libera la memoria reservada.  
end.
```

Listas

Es una colección de nodos. Cada nodo contiene un elemento (valor que se quiere almacenar en la lista) y una dirección de memoria dinámica que indica dónde se encuentra el siguiente nodo de la lista.



HOMOGENEA

Los elementos pueden ser del mismo tipo .

DINÁMICA

El tamaño puede cambiar durante la ejecución del programa.

LINEAL

Cada nodo de la lista tiene un nodo que lo sigue (salvo el último) y uno que lo antecede (salvo el primero).

SECUENCIAL

El acceso a cada elemento es de manera secuencial, es decir, para acceder al elemento 5 (por ejemplo) debo pasar por los 4 anteriores.

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) .

```
Program uno;
```

```
Type
```

```
  listaE= ^datosEnteros; //declaración
```

```
  datosEnteros= record
```

```
    elem:integer;
```

```
    sig:listaE;
```

```
  end;
```

```
Procedure crear (var p: listaE);
```

```

begin
  p:= nil;
end;
Var
  pri: listaE; {Memoria estática reservada}
Begin
  crear (pri);
End.

```

Cargar lista

```

137
138   procedure cargarLista(var pri: lista);
139   var
140     p: persona;
141   begin
142     pri:= nil;
143
144     leerPersona(p);
145     while (p.dni <> 0) do begin
146       agregarAlFinalVl(pri, p);
147       leerPersona(p);
148     end;
149   end;
150
151

```

Recorrido

```

procedure recorrerLista (pI: listaE);
begin
  while (pI <> nil) do
  begin
    write (pI^.elem);
    pI:= pI^.sig;
  end;
end;

```

Agregar al final

Ineficiente

```
22
23 Procedure agregarAlFinalV1 (Var pri: lista; v: integer);
24
25 Var
26   aux, l : lista;
27 Begin
28   new (aux);
29   aux^.dato := v;
30   aux^.sig := Nil;
31
32   if (pri = nil) then
33     pri := aux;
34   else begin
35     l := pri;
36
37     While (l^.sig <> Nil) Then
38       l := l^.sig
39
40     l^.sig := aux;
41   end;
42 End;
43
44
```

pasa por todos los nodos hasta llegar al final y agrega uno al final
Como pri se pasa por referencia copiamos en la variable L que pasa a apuntar a pri y mientras el siguiente de ese L sea distinto de nill, L es el siguiente
si la dirección es nill, lo copio en aux (nodo nuevo)

Eficiente

Mantiene un puntero aparte para el último nodo
evita que tengamos que pasar por todo los nodos

```
46 Procedure agregarAlFinalV2 (Var pri, ult: lista; v: integer);
47
48 Var
49   aux : lista;
50 Begin
51   new (aux);
52   aux^.dato := v;
53   aux^.sig := Nil;
54   If (pri = Nil) Then
55     pri := aux;
56   Else
57     ult^.sig := aux
58     ult := aux;
59 End;
```

Agregar Adelante

```

procedure agregarAdelante (var pI:listaE;num:integer);
Var
    nuevo:listaE;
Begin
    new (nuevo);
    nuevo^.elem:= num;
    nuevo^.sig:=nil;
    if (pI = nil) then
        pI:= nuevo
    else
        begin
            nuevo^.sig:= pI;
            pI:=nuevo;
        end;
End;

```

Agregar atrás

Almacena como parámetro el puntero al primer y al último elemento de la lista

```

procedure agregarAlFinal2 (var
pI,pU:listaE;num:integer);
Var
    nuevo:listaE;
Begin
    new (nuevo);
    nuevo^.elem:= num;
    nuevo^.sig:=nil;
    if (pI = nil) then
        begin
            pI:= nuevo;
            pU:= nuevo;
        end
    end

```



```

else
begin
    pU^.sig:=nuevo;
    pU:= nuevo;
end;
End;

```

Eliminar

```

97
98 procedure eliminarNodo(var pri: lista; num: integer);
99 var
100     ant, act: lista;
101 begin
102     ant:= pri;
103     act:=pri;
104
105     while (act <> nil) and (act^.dato <> num) do begin
106         ant:= act;
107         act:= act^.sig;
108     end;
109
110     if (act <> nil ) then begin
111         if (ant = act) then
112             pri:= act^.sig
113         else
114             ant^.sig := act^.sig;
115
116         dispose(act);
117     end;
118 end;
119

```

Insertar ordenado

depende del dato, se inserta el nodo adelante, atrás o en medio

```

procedure insertar (Var pI: listaE; valor:integer);
Var
    actual,anterior,nuevo:listaE;
Begin
    new (nuevo); nuevo^.elem:= valor; nuevo^.sig:=nil;
    if (pI = nil) then      pI:= nuevo

```

```
    else begin
        actual:= pI; ant:=pI;
        while (actual <> nil) and (actual^.elem <
nuevo^.elem) do
            begin
                anterior:=actual;
                actual:= actual^.sig;
            end;
        end;
    if (actual = pI) then
        begin
            nuevo^.sig:= pI;    pI:= nuevo;
        end
    else
        begin
            anterior^.sig:= nuevo;    nuevo^.sig:= actual;
        end;
    End;
```

MEMORIA

La gestión **de la memoria** es un aspecto fundamental para el correcto funcionamiento de las aplicaciones. Al desarrollar programas, es necesario almacenar y manipular datos en memoria, y la forma en que esta memoria se organiza y asigna tiene un impacto directo en la flexibilidad, eficiencia y complejidad del código.

Aspecto	Memoria dinámica	Memoria estática
Asignación	En tiempo de compilación.	En tiempo de ejecución.
Flexibilidad	Tamaño fijo, no puede cambiar durante la ejecución.	Tamaño ajustable, depende del programador.
Velocidad	Más rápido.	Más lenta debido a la gestión manual.
Liberación	Automática cuando la variable sale de alcance.	Manual, usando Dispose .
Uso típico	Variables locales, globales, arreglos estáticos.	Estructuras complejas como listas, árboles, etc.

¿CÓMO CALCULAR MEMORIA ESTÁTICA Y MEMORIA DINÁMICA?

Memoria estática: consideraremos sólo las **variables locales**, **variables globales de programa** y **constantes**.

Memoria dinámica: consideraremos sólo cuando en la ejecución de un programa se **reserva(NEW)** o **libera memoria(DISPOSE)**.

CADP – MEMORIA DE UN PROGRAMA

```
Program uno;  
Type  
  puntero = ^real;  
  puntero2 = ^char;  
  persona = record  
    nombre:string[20];  
    dni:integer;  
  end;  
  punPer = ^persona;  
Var  
  p,p1:puntero;  
  per: punPer;  
  
Begin  
  new(per);  
  new(p);  
  p^:= 8;  
  p1:= p;  
  dispose(p1);  
End.
```

Clase 8-3

Tabla de ocupación:
char, (1 byte)
boolean, (1 byte)
integer (4 bytes)
real, (8 bytes)
string, (tamaño + 1 byte)
subrango, (depende el tipo)
registro, (suma de sus campos)
arreglos (dimFísica*tipo elemento)
puntero (4 bytes)



CALCULO DE MEMORIA ESTATICA

p = 4 bytes
p1 = 4 bytes
per = 4 bytes
12 bytes

CALCULO DE MEMORIA DINAMICA

new(per) = 25 bytes
new(p) = 8 bytes
dispose(p1) = -8 bytes
25 bytes

Tipo de variable	bytes que ocupa
Char	1 byte
Boolean	1 byte
Integer	6 bytes (varia)
Real	8 bytes (varia)
String	tamaño + 1
Subrango	depende del tipo
Registro	suma de sus campos
Vector	dim física * tipo elem
Puntero	4 bytes

Corrección

¿Cómo defines el concepto de corrección? El **grado** en que una aplicación **satisface las especificaciones** y consigue los **objetivos** encomendados por el cliente.

Técnicas para la corrección de programas:

- **TESTING:** Se analiza el programa con los casos de prueba. Si hay errores se corrigen. Estos dos pasos se repiten hasta que no haya errores.
- **DEBUGGING:** Es el proceso de descubrir y reparar la causa del error. Para esto pueden agregarse sentencias adicionales en el programa que permiten monitorear el comportamiento más cercanamente. Puede tener 3 tipos de errores: sintácticos, lógicos o de sistema.
- **WALKTHROUGH:** Es el proceso de 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 preconcepciones y está predispuesta a descubrir errores u omisiones.
- **VERIFICACIÓN:** Es el proceso de controlar las pre y post condiciones.

Eficiencia

El análisis de la eficiencia de un algoritmo estudia el **tiempo de ejecución** de un algoritmo y la **memoria** que requiere para su ejecución.

Los factores que afectan la eficiencia de un programa

MEMORIA: Se calcula (como hemos visto previamente) teniendo en

cuenta la cantidad de bytes que ocupa la declaración en el programa de:

- constante/s
- variable/s global/es
- variable/s local al programa/es

¿Cómo se miden?

TIEMPO DE EJECUCIÓN: puede calcularse haciendo un análisis empírico o un análisis teórico del programa.

Para medir el tiempo de ejecución se puede realizar un **análisis empírico** o un **análisis teórico**.

- **ANÁLISIS EMPÍRICO**

Requiere la implementación del programa, luego ejecutar el programa en la máquina y medir el tiempo consumido para su ejecución. (depende mucho de la máquina)

- **ANÁLISIS TEÓRICO**

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.

El tiempo de ejecución del **IF**

Tiempo de evaluar la condición + tiempo del cuerpo.

Si hay else, **Tiempo de evaluar la condición + max(then, else).**

El tiempo de ejecución del **FOR**

$(3N + 2) + N(\text{cuerpo del for})$.

N representa la cantidad de veces que se ejecuta el for.

El tiempo de ejecución del **WHILE**

$$C(N+1) + N(\text{cuerpo del while}).$$

N representa la cantidad de veces que se ejecuta el while. ($N \geq 0$)

C cantidad de tiempo en evaluar la condición.

El tiempo de ejecución del **REPEAT UNTIL**

$$C(N) + N(\text{cuerpo del repeat}).$$

N representa la cantidad de veces que se ejecuta el repeat. ($N > 0$)

C cantidad de tiempo en evaluar la condición

A partir de un programa correcto, se obtiene el tiempo teórico del algoritmo y luego el orden de ejecución del mismo. Lo que se compara entre algoritmos es el orden de ejecución.

$$T(n) = 4 \text{ UT}$$

$$O(n) = C \text{ (constante)}$$

$$T(n) = (20 + N) \text{ UT}$$

$$O(n) = N$$

$$T(n) = (20 + \log N) \text{ UT}$$

$$O(n) = \log N$$

$$T(n) = (N * N) = N^2 \text{ UT}$$

$$O(n) = N^2$$

¿Cuál de estos órdenes es más eficiente en tiempo de ejecución?





La eficiencia en tiempo de ejecución de un algoritmo se determina comparando su **orden de ejecución ($O(n)$)**, ya que refleja cómo crece el tiempo requerido en función del tamaño de la entrada (N). **A**

menor tasa de crecimiento de $O(n)$, mayor será la eficiencia del algoritmo.

Orden de eficiencia (de mayor a menor):

1. $O(1)$ (constante, más eficiente, no varía).
2. $O(\log n)$ (logarítmico, crece más despacio si n aumenta.).
3. $O(n)$ (lineal, crece proporcionalmente a n).
4. $O(n^2)$ (cuadrático, menos eficiente).

Orden de eficiencia (de mayor a menor, de más eficiente a menos eficiente):

1. $O(1) \rightarrow$ **IF** (Ejecuta una sola vez, sin importar el tamaño del problema). 
2. $O(\log n) \rightarrow$ **FOR** o **WHILE** si dividen n en cada iteración. 
3. $O(n) \rightarrow$ **FOR**, **WHILE**, **REPEAT UNTIL** en la mayoría de los casos. 
4. $O(n^2) \rightarrow$ Bucles anidados, como **FOR** dentro de otro **FOR** (Ej: algoritmos ineficientes). 

Si hay **bucles anidados**, la eficiencia se degrada rápidamente a $O(n^2)$ o peor.

Si hay **condiciones mal manejadas en WHILE o REPEAT**, puede llegar a ser $O(\infty)$ (bucle infinito).

EXTRAS

1) Eficiencia

a) Definir el concepto

La eficiencia de un algoritmo mide la optimización de los recursos utilizados, como tiempo de ejecución y uso de memoria. Se evalúa a través de la complejidad temporal y espacial, expresadas con notación $O(n)$.

b) ¿Qué se tiene en cuenta para analizar la eficiencia de un algoritmo?

Se consideran principalmente:

- Tiempo de ejecución: número de operaciones realizadas según la entrada.
- Uso de memoria: tanto memoria estática (variables globales y locales) como dinámica (estructuras de datos como listas enlazadas).

c) ¿Cómo se calculan el uso de memoria y el tiempo de ejecución?

- Memoria estática: se calcula sumando el tamaño de todas las constantes, variables y estructuras declaradas.
- Memoria dinámica: depende de las asignaciones (**new**) y liberaciones (**dispose**).
- Tiempo de ejecución: se mide teóricamente asignando una unidad de tiempo a cada operación (asignaciones, comparaciones o condiciones y operaciones aritmética /lógica). Se puede expresar en términos de notación **O(n)**.

d) ¿Toda solución correcta es eficiente?

No. Un algoritmo puede ser correcto pero ineficiente si usa demasiados recursos innecesariamente.

e) ¿Las estructuras de datos elegidas determinan que una solución sea eficiente?

Sí. La elección de estructuras de datos afecta la complejidad computacional.

f) ¿Un programa bien documentado asegura eficiencia?

No. Mejora la legibilidad y mantenimiento del código, lo que puede contribuir indirectamente a optimizarlo.

g) ¿Un programa modularizado asegura eficiencia?

No necesariamente, facilita el desarrollo en equipo, la reutilización del código y el mantenimiento.

2) Modularización y Parámetros

a) Defina modularización y sus ventajas:

La modularización consiste en dividir un programa en módulos más pequeños y específicos. Sus ventajas incluyen:

✓ Facilita la reutilización de código.

- ✓ Mejora la legibilidad y mantenimiento.
- ✓ Permite trabajo en equipo eficiente.
- ✓ Reduce el riesgo de errores al hacer cambios en el programa.

b) ¿Qué módulos dispone Pascal y en qué casos se usan?

- Procedimientos (**procedure**): usados cuando no se requiere devolver un valor o cuando se necesita modificar varias variables a la vez, también realiza acciones sin la necesidad de devolver un resultado.
- Funciones (**function**): siempre devuelven un único valor de un tipo específico.

Diferencias clave:

1. Cantidad de resultados: una función devuelve un solo valor, un procedimiento puede modificar múltiples o no tener un resultado.
2. Forma de operación interna: una función usa una única variable de retorno, mientras que un procedimiento trabaja con parámetros.
3. Cómo se invocan: una función se puede usar dentro de una expresión (**x := f(a)**), mientras que un procedimiento se llama como una instrucción independiente (**p(a)**).

c) ¿Todo **procedure** se puede escribir como una **function**?

No siempre. Un **procedure** puede modificar varias variables o realizar acciones sin devolver un resultado, lo que no encaja en una función.

d) ¿Siempre se puede transformar una **function** en un **procedure**?

Sí, asignando el resultado a un parámetro por referencia.

e) ¿Cómo se comunican los módulos?

A través de variables globales (poco recomendable) o parámetros

(mejor opción porque mantiene la integridad y seguridad de los datos).

f) ¿Qué es el alcance de una variable?

Es el rango dentro del cual se puede acceder a una variable:

- Global: accesible desde cualquier parte del programa.
- Local: solo dentro del procedimiento o función donde fue declarada.
- Parámetros: su alcance depende de cómo se pasen (por valor o referencia).

g) Diferencias entre variable global y local.

- Global: accesible desde cualquier módulo, pero aumenta el riesgo de errores y consumo innecesario de memoria.
- Local: solo se usa en el bloque donde se declara, lo que evita conflictos y facilita el mantenimiento.

h) ¿Qué es un parámetro en la modularización?

Es una variable que se usa para comunicar información entre módulos, evitando el uso de variables globales.

i) Diferencias entre parámetros por valor y por referencia

- Por valor: el módulo recibe una copia del dato, por lo que los cambios no afectan al original.
- Por referencia: se pasa la dirección de memoria, por lo que cualquier cambio afecta directamente la variable original.

j) ¿Por qué son útiles los parámetros en la modularización?

Porque permiten la comunicación entre módulos sin depender de variables globales, mejorando la claridad y seguridad del código.

k) ¿Qué es la reusabilidad en la modularización?

Es la capacidad de utilizar los mismos módulos en diferentes partes del programa o en otros programas, reduciendo la redundancia y facilitando el mantenimiento.

3) Estructuras de Datos

a) Definición

Una estructura de datos es una forma de organizar y almacenar datos en memoria para facilitar su uso y optimizar su manipulación mediante operaciones específicas.

b) Criterios de clasificación de las estructuras de datos

Las estructuras de datos varían según los siguientes criterios:

Criterio	Opciones	Ejemplos
Homogeneidad	Homogénea (mismo tipo de dato) / Heterogénea (distintos tipos de datos)	Vectores (homogéneos), Registros (heterogéneos)
Acceso	Secuencial / Directo (índice)	Listas enlazadas (secuencial), Vectores (directo)

Linealidad	Lineal / No lineal	Vectores y Listas (lineales), Árboles y Grafos (no lineales)
Tamaño	Estático / Dinámico	Vectores (estático), Listas enlazadas (dinámico)

b2) Diferencias entre registro, vector y lista enlazada

- Registro: estructura heterogénea, de tamaño fijo, donde los datos se organizan en campos con diferentes tipos.
- Vector: estructura homogénea, estática, con acceso directo mediante índice, y lineal porque se conoce el orden de los elementos.
- Lista enlazada: estructura homogénea, de tamaño dinámico, con acceso secuencial y elementos enlazados mediante punteros. Pueden no ocupar memoria contigua.

c) Inserción de un elemento en un vector y una lista

- Vector:
 1. Verificar si hay espacio disponible.
 2. Desplazar los elementos hacia la derecha para hacer espacio (si se inserta en una posición intermedia).
 3. Insertar el nuevo elemento en la posición deseada.
 4. Incrementar la dimensión lógica.
- Lista enlazada:
 1. Crear un nuevo nodo en memoria.
 2. Asignar el valor al nuevo nodo.
 3. Ajustar los punteros para enlazar el nuevo nodo en la posición deseada.

d) Análisis del tiempo de ejecución

- Vector:
 - Mejor caso ($O(1)$) → Si se inserta al final.
 - Peor caso ($O(n)$) → Si se inserta al inicio y hay que desplazar todos los elementos.
- Lista enlazada:
 - Siempre $O(n)$ en acceso secuencial, ya que es necesario recorrer la lista para encontrar la posición correcta.
 - Inserción en cualquier parte $O(1)$ si se tiene la referencia directa.

e) Diferencia entre dimensión lógica y física

- Dimensión física: tamaño total reservado en memoria. Ejemplo: un vector con capacidad para 100 elementos.
- Dimensión lógica: cantidad de elementos efectivamente ocupados. Ejemplo: si un vector de capacidad 100 tiene solo 20 elementos, su dimensión lógica es 20.

Se usa para optimizar el almacenamiento y controlar el acceso a los datos.

f) Diferencia entre **DISPOSE** y **NIL**

- **DISPOSE(p)**: libera la memoria asignada al puntero **p**, permitiendo su reutilización.
- **p := NIL**: corta la referencia del puntero, pero no libera la memoria.

Si no se usa **DISPOSE**, se pueden generar fugas de memoria.

g) Riesgos de usar variables globales





- Puede generar errores difíciles de depurar si es modificada por varios módulos.
- Reduce la claridad del código, ya que no es evidente qué módulos la modifican.
- Dificulta la reutilización de los módulos en otros programas.

g2) Ventajas de usar parámetros por referencia en lugar de variables globales

- Se reduce la cantidad de identificadores en el programa.
- Se mejora la claridad y seguridad, ya que el flujo de datos es explícito.
- Permite modularizar de forma efectiva sin depender de estados globales.

h) Operaciones con registros

Si **A** y **B** son registros de la misma estructura:

Operación	¿Es válida?	Justificación
A := B	 No	En Pascal, no se pueden asignar registros completos, se debe hacer campo por campo.
A = B	 No	No se pueden comparar registros completos, solo campo por campo.
READ(A)	 No	Se debe leer campo por campo.
WRITE(B)	 No	Se debe escribir campo por campo.

i) Algorítmica de corte de control

Se usa cuando una estructura está ordenada por un criterio específico, permitiendo:

- ✓ Recorrerla eficientemente sin crear estructuras auxiliares.
 - ✓ Procesar datos relacionados (por ejemplo, sumar ventas por cliente en un listado ordenado).
-

4) Otras preguntas

a) Importancia de que el lenguaje permita tipos de datos definidos por el usuario

- Permite estructurar mejor la información, facilitando la programación modular.
- Mejora la legibilidad y mantenimiento del código.
- Facilita la reutilización y adaptación a diferentes problemas.

Ejemplo en Pascal:

pascal

CopiarEditar

type

Alumno = record

Nombre: string;

Edad: integer;

end;

b) Diferencia entre un lenguaje "fuertemente tipado" y uno "dinámicamente tipado"

Tipo

Características

Ejemplo de
lenguaje

Fuertemente tipado	Requiere declarar el tipo de cada variable y respeta la asignación de tipos estrictamente.	Pascal, Java, C++
Dinámicamente tipado	No es necesario declarar tipos, y las variables pueden cambiar de tipo en tiempo de ejecución.	Python, JavaScript

c) ¿Toda solución es correcta?

No necesariamente. Se evalúan dos aspectos:

1. Corrección: el algoritmo debe producir resultados esperados para todas las entradas posibles.
2. Eficiencia: aunque sea correcto, si usa demasiados recursos, puede no ser práctico en la realidad.

Ejemplo:

Un algoritmo de búsqueda en un vector desordenado es correcto si encuentra el elemento, pero no es eficiente ($O(n)$). En cambio, una búsqueda binaria en un vector ordenado ($O(\log n)$) es más eficiente.