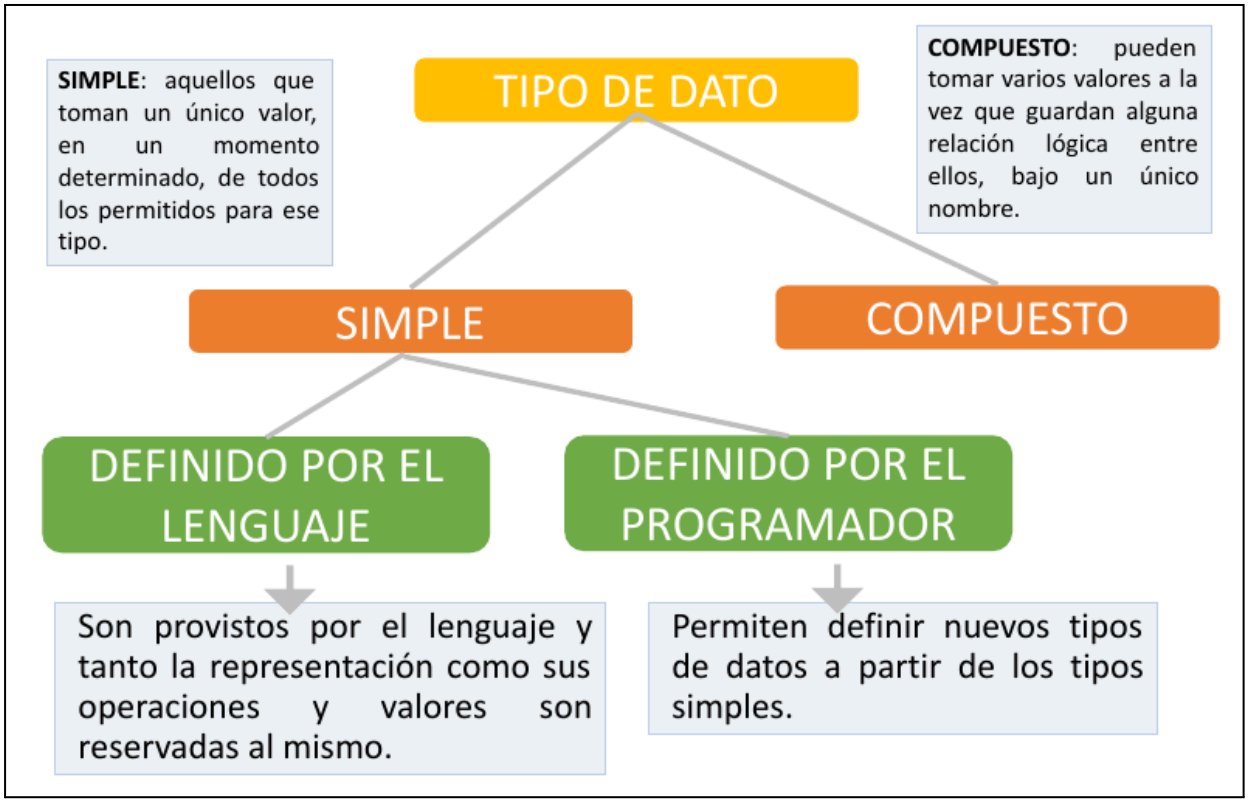
Conceptos de algoritmos, datos y programas

Definiciones preliminares

La **informática** es la ciencia que estudia el análisis y la resolución de problemas utilizando computadoras. Su objetivo es resolver problemas del mundo real utilizando una computadora, o, dicho más precisamente, utilizando un software. En la cursada de CADP, los problemas son los ejercicios de las prácticas, y las soluciones son los programas que hacemos en Pascal.

Un **dato** es una representación de un objeto del mundo real mediante la cual podemos modelizar aspectos del problema que se quiere resolver con un programa sobre una computadora. Puede ser constante, como el valor de π, o variable, como el valor de un contador.

Ahora bien, no todos los datos son iguales, sino que existen distintos tipos de datos. Acá tenemos una visión general e inicial:



Pero para que todo esto tenga sentido, veamos directamente cómo se aplica en el lenguaje que vamos a utilizar: Pascal.

Pascal es un lenguaje de programación creado por el profesor suizo Niklaus Wirth y publicado en 1970. Este lenguaje se caracteriza por dos aspectos fundamentales que lo distinguen:

1) **Tipado Fuerte**. Pascal es un lenguaje fuertemente tipado (*strongly typed*), lo que significa que:
- Todas las variables deben ser declaradas con un tipo específico antes de ser utilizadas
- El compilador verifica que cada variable se use de manera consistente con su tipo declarado
- No se permiten conversiones automáticas entre tipos incompatibles, evitando errores comunes de programación

2) **Programación Estructurada**. Pascal fue diseñado para promover la programación estructurada (*structured programming*), caracterizada por:
- La organización del código en funciones y procedimientos claramente definidos
- El uso de estructuras de control bien definidas (if-then-else, while, for, case)
- La eliminación de saltos incondicionales (goto) que hacían difícil seguir el flujo del programa
- La facilidad para leer y mantener el código resultante

El primer aspecto mencionado es fundamental: *el tipo de dato de todas las variables debe ser declarado previamente para que su uso quede habilitado*. Si vamos a declarar una variable no podemos poner solamente su nombre. Tenemos que darle un nombre y aclarar de qué tipo de dato se trata.

Este paso es muy importante, ya que el tipo de dato determina cuáles operaciones son válidas (y cuáles no) sobre esa variable; y una vez que definimos el tipo de dato, no podemos modificarlo durante la ejecución del programa –porque Pascal es un lenguaje con tipado estático [*static typing*], no dinámico.

Ahora sí, veamos algunos casos concretos.

Para los números enteros tenemos el **integer**, un tipo de dato simple y ordinal. La definición de simple ya la vimos en el cuadro anterior. Pero, ¿ordinal?

> En programación, un tipo de dato ordinal es un tipo de dato con la propiedad de que sus valores pueden ser contados. Es decir, que sus valores pueden colocarse en una correspondencia uno a uno con los números enteros positivos.

En otras palabras, un tipo de dato es ordinal si los podemos contar y ordenar de menor a mayor de forma sencilla. Esto sucede, obviamente, con los integer, y también con otros tipos de datos que veremos más adelante.

El caso de los enteros. ¿Por qué funciona aquí? Porque naturalmente, si yo elijo un número entero cualquiera como punto de partida, puedo ordenarlos, a partir de ahí, de forma sencilla

poniéndolos en una correspondencia uno a uno con los enteros positivos. Supongamos que elijo el -2 como primer número, entonces sé que después del -2 está el -1, después el 0, el 1, etc., *y sé también que no hay nada en el medio de cada uno de ellos*. De esta manera, podemos llegar fácilmente a la correspondencia mencionada:

Enteros positivos	1	2	3	4	5	…
Subrango de integer	-2	-1	0	1	2	…

El caso de los caracteres. Obviamente son simples, y también ordinales, porque —siguiendo la tabla ASCII— podemos ponerlos de forma ordenada en una correspondencia uno a uno con los enteros positivos:

Enteros positivos	1	2	3	4	5	…
Caracteres	a	b	c	d	e	…

El caso boolean, que solo tiene dos valores posibles.

Enteros positivos	1	2
Boolean	true	false

Ahora bien, si trato de hacer esto con los números reales, resulta imposible, porque entre dos reales cualesquiera, hay infinitos reales en el medio. Con lo cual, no sé cual es el siguiente ni el anterior de un real. Pensemos por ejemplo cuál sería el que le sigue al 1. Podría ser 1.1, o 1.0005432, o cualquier otro entre 1 y 2. En otras palabras, no podemos establecer una correspondencia uno a uno con los enteros positivos. Ergo: los números reales no son un tipo de dato ordinal.

Pero antes de enloquecer como Cantor, volvamos al humilde y sencillo integer, y veamos qué podemos hacer con él. Al integer podemos aplicarle los cuatro operadores matemáticos principales (+, -, * y /), los operadores lógicos (=, <>, <, >, <=, >=), y los operadores enteros mod y div.

Con la suma, la resta, la multiplicación y los comparadores no hay mucho misterio. Con la división vale aclarar dos cosas:

- Tanto en la vida real como en Pascal no se puede dividir por 0. Si eso sucede, el programa crashea.
- El resultado de la división entre dos números enteros puede no ser un entero, así que si bien podemos dividir dos integer, el resultado no puede asignársele a una variable de tipo integer. Sí podemos almacenarlo en un *real*, o directamente imprimirlo.

¿Y los operadores enteros mod y div? Mod sirve para quedarnos con el resto de una división, y div con el cociente.

Ejemplo: si dividimos $\frac{23}{7}$ con una calculadora (o en Pascal, usando el operador /), nos da un número como 3.285714286. Pero si trabajamos con los operadores enteros, sólo obtendremos números enteros, como cuando hacemos la división en papel y no nos molesta ignorar el resto. Por eso, este programa

```
program enteros;
var a, b : integer;
begin
   a := 23;
   b := 7;
   writeln(a div b);
   writeln(a mod b);
end.
```

imprime lo siguiente:

```
3
2
```

Y no está de más recordar que todo esto viene del teorema de la división, según el cual $m = qd + r$. Y por eso obtenemos que $23 = 7 \cdot 3 + 2$.

Para los números reales, tenemos el tipo de dato **real**, que es simple pero no ordinal, y nos permite usar los mismos operadores que antes, excepto, obviamente, los operadores enteros (div y mod).

Para cuando queremos decir simplemente si algo es verdadero o falso, tenemos el tipo de dato lógico o **boolean**, que es simple y ordinal.

Con respecto a las operadores lógicos, sólo usaremos el **and**, el **or** y el **not**. Estos son muy importantes al momento de escribir las condiciones.

Para los caracteres individuales, tenemos el **char**, que es simple y ordinal. Permite usar los operadores lógicos (=, <>, <, >, <=, >=).

Para las cadenas de caracteres (hasta 255 según internet, hasta 256 según la cátedra), tenemos el **string**, que es compuesto[1]. Permite usar los operadores lógicos (=, <>, <, >, <=, >=). Cabe aclarar que si bien Pascal no es *case sensitive* con las instrucciones del programa, sí lo es con

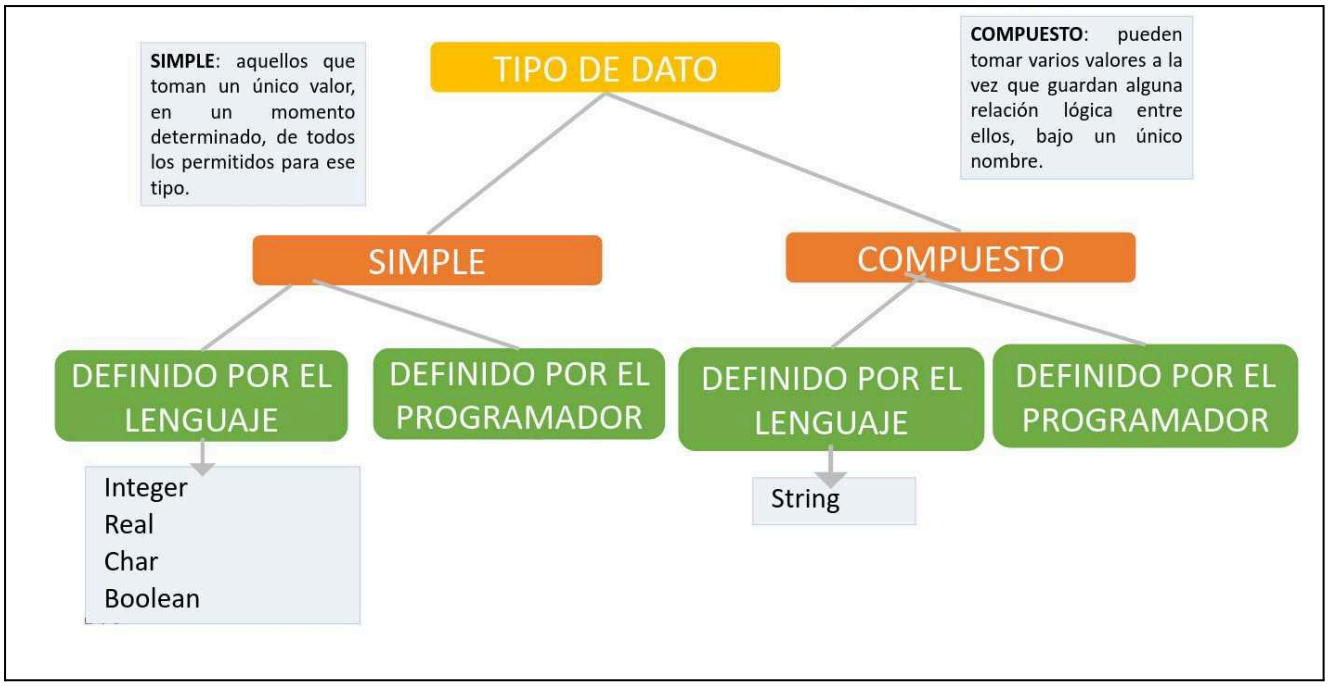
[1] El string, ¿es un tipo de dato ordinal? La cátedra no lo dice. Dependiendo del contexto, puede serlo o no. En Pascal, por lo que sé, podríamos considerarlo ordinal, ya que el string tiene un límite de caracteres y Pascal utiliza por defecto el orden alfabético, entonces fácilmente se puede decir cuál es el string que le sigue a otro. Sin embargo, no es algo que hayamos visto mucho en la materia y conviene no meterse demasiado.

los chars o los strings (porque lo que en verdad compara son los números de la tabla ASCII). Así que el string ‘hola’ y el string ‘Hola’, para Pascal, son distintos.

Ahora bien, cada uno de estos tipos de datos podemos usarlo en una constante o en una variable, dependiendo de las necesidades del programa.

Tanto una variable como una constante puede entenderse como una zona de memoria cuyo contenido va a ser alguno de los tipos mencionados anteriormente, donde la dirección inicial de esta zona se asocia con el nombre de la variable. La única diferencia entre las dos, es que la primera puede cambiar su valor durante la ejecución del programa, mientras que la segunda no.

Mapa conceptual de los tipos de datos vistos hasta ahora:



Dos conceptos poco usados: **precondición** y **postcondición**. La *precondición* es la información que se conoce como verdadera antes de iniciar el programa (o módulo), mientras que la *postcondición* es la información que debería ser verdadera al concluir el programa (o módulo), si se cumplen adecuadamente los pasos especificados.

Veamos dos operaciones fundamentales para el input y el output de los programas: las operaciones de lectura y de escritura.

El **read()** o readln(), sirve para que el usuario ingrese datos. Es importante que los datos que ingresa sean del mismo tipo que los especificados por la variable del read(), ya que, de no hacerlo, el programa puede crashear durante la ejecución. (Por ejemplo, si tenemos una

variable num1 de tipo integer, hacemos read(num1) y el usuario en vez de ingresar simplemente un número entero ingresa ‘7g’, el programa muere).

El **write()** o writeln(), sirve para mostrar o imprimir información en pantalla. Puede mostrar una string de forma directa, o desde una variable. Tal como se puede ver en el siguiente ejemplo:

```
program palabras;
var palabra : string;
begin
   palabra := 'Ah re loco';
   writeln(palabra);
   writeln('Mal loco');
end.
```

El cual imprime en la terminal:

```
Ah re loco
Mal loco
```

También se pueden combinar distintas cosas dentro del write() usando muchas comas y comillas, de este manera:

```
program palabras_y_mas;

var materia : string;
   num : integer;

begin
   materia := 'EPA';
   num := 2;
   writeln('Me saqué un ', num, ' en ', materia);
end.
```

Que imprime:

```
Me saqué un 2 en EPA
```

Estructuras de control

En pocas palabras, un programa informático es un conjunto ordenado de instrucciones escritas en un determinado lenguaje de programación con el objetivo de resolver algún problema. Y podríamos decir que sus componentes principales son los datos, los operadores y las estructuras de control. Los datos, básicamente, almacenan la información; los operadores la transforman o realizan algún análisis; y las estructuras de control se encargan de dirigir el rumbo del programa: indican cuál va a ser la próxima instrucción a ejecutar.

Aquí veremos cinco tipos de estructura de control: secuencia, decisión, selección, iteración y repetición.

1. Secuencia

Se trata de la estructura de control más simple. Está representada por una sucesión de operaciones, en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones (leyendo de arriba hacia abajo y de izquierda a derecha).

Podríamos decir que es la estructura de control inherente al programa (al menos a Pascal), y que no debemos hacer nada para que esté presente, porque es la estructura que está por defecto. A menos que le indiquemos otra cosa, el programa siempre ejecutará las instrucciones de esta manera. Pero obviamente no siempre vamos a querer ejecutar las instrucciones secuencialmente, y por eso están las demás estructuras.

2. Decisión

A veces vamos a necesitar que se ejecuten algunas instrucciones si sucede algo, y otras (o ninguna) si eso no sucede. Para eso está el **if**, que puede estar o no acompañado por el **else**.

Es una estructura muy simple que no requiere mucha explicación, aunque sí dos aclaraciones:

- En inglés, “if … then …, else …”, significa “si … entonces …, si no …”.
- En Pascal puede escribirse de distintas maneras según la cantidad de instrucciones que tenga, y siempre hay que tener cuidado con los punto y coma. Tanto si estamos en la parte del *if* como en la del *else*, si tenemos una sola instrucción podemos no escribir el *begin… end*, pero si tenemos más de una es obligatorio (o al menos muy conveniente).
- Obviamente, pueden anidarse (como prácticamente todo).

Aquí van algunos ejemplos:

```
if (...) then
   instrucción1;

if (...) then
   instrucción1
else
   instrucción1;

if (...) then
begin
   instrucción1;
   instrucción2;
   ...
   instrucciónN;
end;

if (...) then
begin
   instrucción1;
   instrucción2;
   ...
   instrucciónN;
end
else
begin
   instrucción1;
   instrucción2;
   ...
   instrucciónN;
end;
```

3. Selección

Esta estructura permite realizar distintas acciones dependiendo del valor de una variable de tipo simple y ordinal. Básicamente, sirve para cuando tenemos muchas opciones y queremos elegir una sola. Además, puede simplificar mucho la escritura de algunos programas.

Ejemplo: supongamos que queremos hacer un programa que lea un char y diga si es una letra minúscula, mayúscula, un dígito o un caracter especial. Usando distintos if, nos constaría bastante trabajo. En cambio, con el **case**, nos queda una cosa así:

```
program bienvenidoCASE;
var car : char;
begin
   readln(car);
   case car of
       'a'..'z': writeln('minúscula');
       'A'..'Z': writeln('mayúscula');
       '0'..'9': writeln('dígito');
       else writeln('especial');
   end;
end.
```

Y al igual que antes, si queremos ejecutar más de una instrucción en algún caso, debemos agregar un begin al principio y un end al final. De esta manera:

```
   case car of
       'a'..'z': begin
                   writeln('minúscula');
                   writeln('ejemplo');
                 end;
       'A'..'Z': writeln('mayúscula');
       '0'..'9': writeln('dígito');
       else writeln('especial');
   end;
```

4. Iteración

Puede ocurrir que se desee ejecutar un bloque de instrucciones desconociendo el número exacto de veces que se ejecutan. Y para eso tenemos dos opciones: el **while** y el **repeat**.

El while se considera de tipo pre condicional, ya que primero pregunta si se cumple la condición, y en función de eso ejecuta o no el código que le fue asignado. Por eso su bloque de instrucciones puede ejecutarse 0, 1 o más veces. Se escribe así:

```
while (esta condición es verdadera) do // mientras (esto sea verdad) hacer
begin
   ...
end;
```

Nuevamente, el begin y el end sólo es necesario cuando hay más de una instrucción.

El repeat, en cambio, es post condicional: primero ejecuta el bloque una vez, y luego pregunta para saber si debe repetir otra vez el bloque o no. Esto significa que siempre se ejecuta al menos una vez. Es decir, su bloque de instrucciones se ejecuta 1 o más veces.

```
repeat // repetir
   ... // estas instrucciones
until (esta condición sea verdadera); // hasta que esto sea verdad
```

En el repeat nunca hace falta poner *begin… end*.

En CADP, normalmente lo importante es reconocer lo siguiente: si queremos leer datos hasta que se ingrese uno que no debe procesarse, usamos el while, y si el último debe procesarse, usamos el repeat.

5. Repetición

Curiosamente, la estructura de control para realizar repeticiones no es el repeat, sino el **for**.

Cuando sabemos exactamente cuántas veces vamos a ejecutar cierto bloque de instrucciones, usamos el *for*, así:

```
for i := 1 to 10 do
begin
   ...
end;
```

Al igual que antes, si hay una sola instrucción en el bloque, no es necesario agregar begin y end.

Algunas cosas a destacar del for:

- Nos ofrece un índice (en este ejemplo, la variable *i*) que podemos consultar durante la ejecución, aunque no lo podemos modificar. De hecho, el compilador no permite que a la variable que hace de índice se le asigne ningún valor adentro del bloque del *for*, ni siquiera el propio valor que ya posee. Dicha variable se incrementa o se decrementa de forma automática y no la podemos tocar.
- Para que vaya decrementando, lo escribimos así:

```
for i := 10 downto 1 do
begin
   ...
end;
```

- Podemos usar variables para definir de dónde a donde debe repetirse. Así:

```
program probando_el_for;  // Imprime todos los números entre num1 y num2,
                          // siempre y cuando num1 <= num2
var i, num1, num2 : integer;

begin
   readln(num1);
   readln(num2);
   for i := num1 to num2 do
       writeln(i);
end.
```

Si ingresamos el 2 y el 5, imprimirá

```
2
3
4
5
```

Pero si ingresamos 5 y 2, no imprimirá nada, ni tampoco dará error. Esto significa que el for puede ejecutarse 0, 1 o más veces. Ahora bien: el ejemplo oficial de que puede ejecutarse 0 veces, es cuando tenemos un for para recorrer dede 1 hasta dimL (o sea, la dimensión lógica de un array), y el array está vacío, con lo cual dimL vale 0.

¿Y cómo sabemos cuántas veces se va a repetir? Si tenemos la clásica expresión que dice que va desde 1 hasta n, así:

for i := 1 to n do // siendo n algún entero positivo, como por ejemplo 10

es obvio que se va a repetir n veces.

Pero, si dice algo como

for i := 4 to 17 do

¿cómo hacemos?

Hay dos maneras. Mi recomendación es llevarlo a la expresión estándar, es decir, llevar el límite inferior a 1 y ver dónde queda el otro (cuando le aplicamos la misma operación). En este caso, para llevar *i* a 1 habría que hacer $4 - 3 = 1$. Y lo que le hayamos hecho al límite inferior, hay que hacérselo al superior también, entonces hacemos $17 - 3 = 14$. Con lo cual, sabemos que el bloque del *for* se ejecutará 14 veces.

La propuesta de la cátedra es hacer la cuenta $límite\ superior - límite\ inferior + 1$, que en este caso sería $17 - 4 + 1 = 14$.

Máximos y mínimos

Todo ser humano, en algún momento de su existencia, necesitará calcular máximos y mínimos. Hay varias maneras de hacerlo, pero en CADP hay un par de soluciones estándar que conviene seguir.

Básicamente, si tenemos un solo máximo (o un solo mínimo), lo que hay que hacer es crear una variable para almacenar ese valor, inicializarla con un valor muy bajo o muy alto (según el caso), y luego compararla con los valores que corresponda, y actualizarla en caso de ser necesario.

Ejemplo: leer 10 números enteros e imprimir el más alto.

```
program maximos_a_full;

var i, num, max : integer;

begin
   max := -999; // Inicializamos max en algún valor muy bajo
   for i := 1 to 10 do
   begin
       readln(num);
       if num > max then // ¿El número leído es mayor al máximo almacenado?
           max := num; // Si es mayor, lo actualizamos
   end;
   writeln('El número más grande leído fue el ', max);
end.
```

También puede suceder que no tengamos que imprimir el número máximo, sino algún dato asociado a él, como en el siguiente caso.

Ejemplo: realizar un programa que lea número de alumno y promedio hasta leer un promedio igual a 0. Al finalizar informar el alumno con el promedio más alto.

```
program promedios_de_los_pibes;
// Se asume que se ingresará al menos un promedio válido

var prom, max_prom : real;
   alu, max_alu : integer;

begin
   max_prom := -1; // Inicializamos max_prom en algún valor muy bajo
```

```
   write('Ingrese el número de legajo: '); readln(alu);
   write('Ingrese el promedio: '); readln(prom);
   while (prom <> 0) do
   begin
       if (prom > max_prom) then // Estos valores son los que se comparan
       begin
           max_prom := prom;
           max_alu := alu; // Estos, los que nos interesa conocer al final
       end;
       write('Ingrese el número de legajo: '); readln(alu);
       write('Ingrese el promedio: ');readln(prom);
   end;
   writeln('El legajo del alumno con el promedio más alto es ', max_alu);
end.
```

Recordatorio: para calcular el máximo y el mínimo en el mismo ejercicio, hay que preguntar por separado. No podemos usar un *if*... *else*... porque puede generar problemas. Hay que usar un *if* para el máximo, y un *if* para el mínimo.

Veamos ahora qué sucede si queremos saber dos máximos o dos mínimos.

Supongamos que queremos encontrar dos mínimos. La declaración es igual que antes: necesitamos una variable para el mínimo más mínimo, y otra para el segundo. Convencionalmente, las llamamos min1 y min2.

La inicialización es distinta: no hace falta inicializar las dos. Con inicializar min1 alcanza, ya que en la primera comparación, lo primero que se va a actualizar es el min1.

La estructura es un poco distinta, porque hay tres opciones: el nuevo valor puede ser menor a min1, en cuyo caso el valor que estaba en min1 pasa a min2, y el nuevo queda en min1; puede ser únicamente menor a min2, en cuyo caso se actualiza solamente min2; y puede ser que no sea menor a ninguno, entonces no hay que actualizar nada.

Normalmente preguntamos en ese orden, y queda una estructura con este estilo:

```
   if (num < min1) then
   begin
       min2 := min1;
       min1 := num;
   end
   else
   begin
       if (num < min2) then
           min2 := num;
   end;
```

Inventando tipos de datos: el subrango

A veces los tipos de datos de Pascal nos quedan cortos. Queremos otros tipos de datos que sean más adecuados a nuestras necesidades. Para eso está **type**, el espacio donde Pascal nos deja definir nuevos tipos de datos a partir de los ya existentes.

Uno de los más comunes es el subrango. Es un tipo ordinal que consiste de una sucesión de valores de un tipo ordinal (predefinido o definido por el usuario) tomado como base. Las operaciones permitidas sobre este nuevo tipo, dependen del tipo de dato tomado como base.

Hacer un subrango resulta útil cuando sabemos exactamente el rango de los valores que vamos a usar. Como por ejemplo, para cuando queremos tener una variable numérica para los días de la semana, sabemos que van a ser 7. Entonces podríamos armar algo como esto:

```
program eight_days_a_week;

const ... // Acá irían las constantes, si las hubiera

type  dias = 1..7; // <-- Acá los nuevos tipos de datos

// Acá los módulos, si los hubiera

var dia : dias; // Variables del programa principal

begin
   dia := 2;
   writeln(dia);
end.
```

Modularización

Cuando los programas tienen menos de veinte líneas, la modularización parece inútil –y tal vez lo sea en esos casos. Pero cuando tenemos programas más grandes, resulta casi indispensable.

La modularización no es más que dividir el problema en partes independientes. Pero esto es fundamental, ya que conlleva muchas ventajas, debido a que aumenta

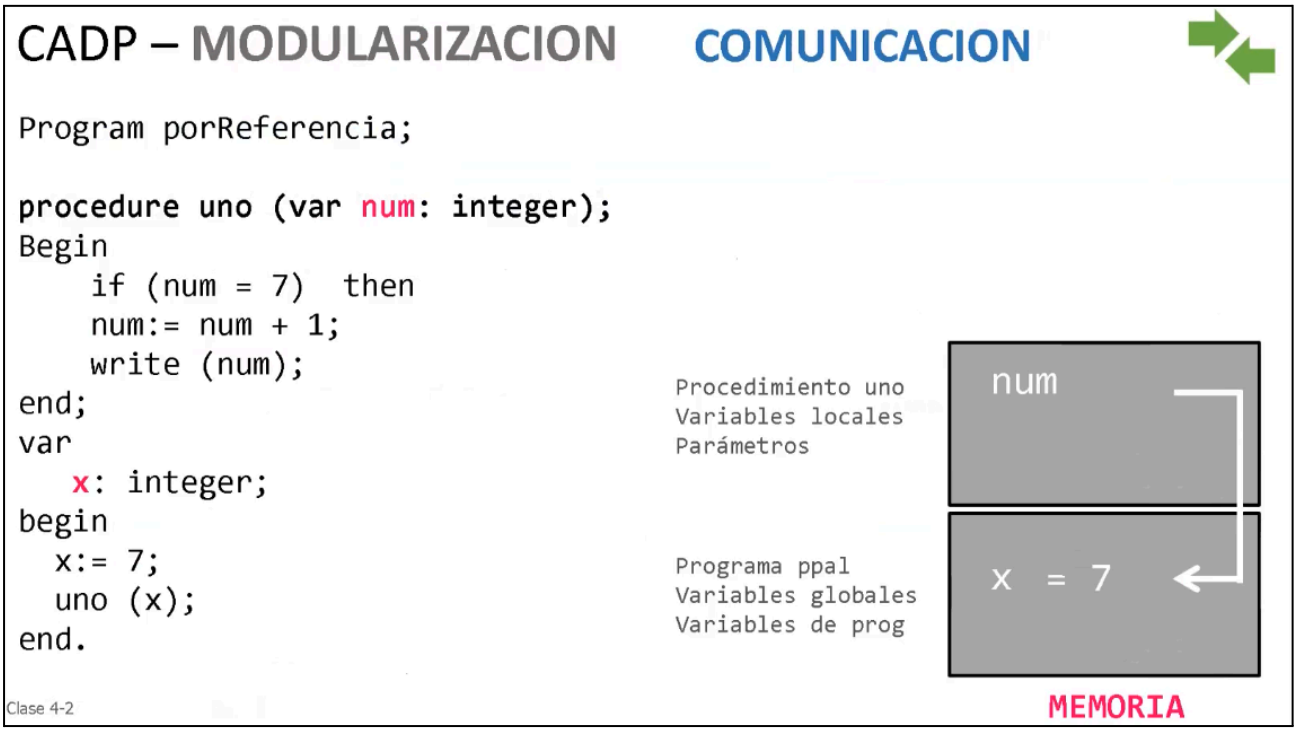
- la legibilidad del código (porque es más facil leer y entender el código en partes antes que todo junto)
- la productividad (porque, en un equipo de trabajo, diferentes programadores pueden trabajar con diferentes módulos al mismo tiempo)

- la reusabilidad (porque un módulo hecho para un programa puede llegar a servir para varios otros programas)
- la facilidad de mantenimiento (por todo lo anterior, es obvio que es más fácil modificar y corregir un programa modularizado: es más fácil encontrar el error, es más fácil determinar en qué sector deben cambiarse cosas y dónde no, etc.)
- la facilidad de crecimiento (ya que es más cómodo andar agregando y sacando módulos que pedazos de código del programa principal)

Tenemos dos tipos de módulos: los procedimientos y las funciones.

La principal diferencia es que el *procedure()* puede retornar 0, 1 o más datos, mientras que la *function()* siempre debe retornar exactamente 1 dato.

Al *procedure()* podemos pasarle datos mediante parámetros. Estos parámetros pueden ser por valor o por referencia. Si es por valor, se hace una copia del dato, y por más que se modifique adentro del procedimiento no va a devolverlo cambiado. En cambio, si lo pasamos por referencia (utilizando la palabra clave *var*) entonces se va a modificar también. Internamente, lo que sucede es que el parámetro por referencia no es otra variable aparte con otra información, sino que es un puntero que apunta a la misma posición que la variable enviada como parámetro. Ilustración:



A la *function()* sólo podemos pasarle parámetros por valor, y al final de la misma debería haber una línea con el nombre de la función y un := que especifique qué es lo que va a devolver.

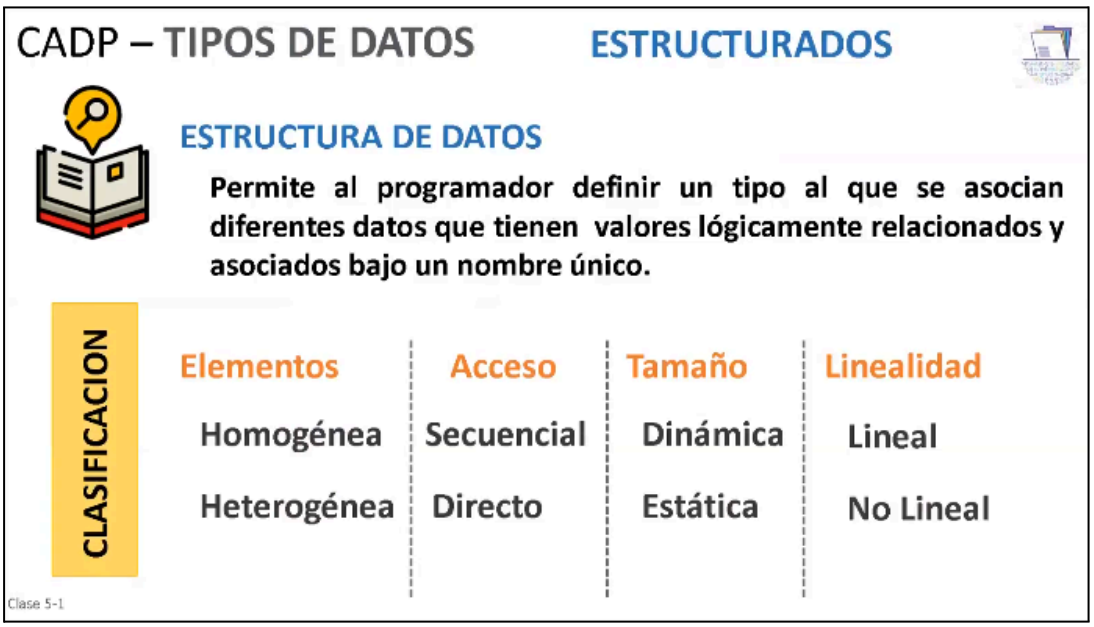
Pero los módulos no sólo usan parámetros, también pueden crear sus propias variables locales. Entonces, si tenemos dos variables con el mismo nombre, ¿cómo sabemos a cuál se está haciendo referencia? Para saberlo, hay que seguir "la jerarquía de variables", es decir, el orden en el cuál pregunta por dónde está la variable que va a usar. Dicho orden es el siguiente:

1) variables locales
2) parámetros
3) variables globales (y/o constantes)

De todas formas, obviamente no es recomendable usar el mismo nombre para todo, ni tampoco usar variables globales.

Por otro lado, cabe resaltar que este tema se relaciona también con el alcance de las variables. Obviamente, si yo declaro una variable local adentro de un procedimiento, no podré usarla en el programa principal por ejemplo. Lo mismo sucede con los procedimientos y los tipos de datos. Pueden ser declarados adentro de otros procedimientos, pero en este caso no estamos ante una buena práctica, ya que no podremos usarlos en otras partes del programa.

Tipos de estructuras de datos



Si los elementos que componen la estructura son —necesariamente— todos del mismo tipo, entonces la estructura es de tipo **homogénea**. De lo contrario, si los elementos que la componen *pueden* ser de distinto tipo, entonces es **heterogénea**.

Si su tamaño se define cuando se declara la estructura y no puede variar durante la ejecución del programa, entonces es de tipo **estática** (o sea, se calcula cuando se compila el programa y no varía). En cambio, si puede variar con la ejecución del programa, es **dinámica**.

Si para acceder a un elemento particular se debe respetar un orden predeterminado, o sea, siguiendo una secuencia ordenada hasta llegar a donde queremos, entonces la estructura tiene un tipo de acceso **secuencial**. En contraste a esto, si podemos acceder directamente a donde se nos antoje sin necesidad de andar recorriendo los elementos anteriores (por ejemplo, referenciando una posición), entonces la estructura tiene un tipo de acceso **directo**.

Por último, una estructura es **lineal** cuando está formada por 0, 1 o varios elementos que guardan una relación de adyacencia ordenada, donde a cada elemento le sigue y le precede uno solamente. Y es **no lineal** si para un elemento dado pueden existir 0, 1 o más elementos que le suceden y 0, 1 o más elementos que le anteceden.

Veamos ahora cómo se clasifican los últimos tipos de datos que trabajamos en la materia, que fueron los siguientes cuatro: registros, vectores, punteros y listas.

El **REGISTRO** es un tipo de dato estructurado, que permite agrupar diferentes clases de datos en una estructura única bajo un único nombre. Cada uno de los elementos/datos que lo componen se llaman “campos”. El registro es una estructura de tipo heterogénea y estática[2].

El ARRAY (o arreglo) es una estructura de datos compuesta que permite acceder a cada elemento mediante una variable índice (o algunas variables índices), que referencia(n) la posición del elemento dentro de la estructura de datos. El array puede ser de *n* dimensiones, es decir que nos permite crear matrices. Sin embargo, no veremos eso en este curso. Aquí trabajaremos con arreglos de una dimensión, es decir, con vectores (aunque, por costumbre, también los llamamos arreglos). Un **VECTOR** es una colección de elementos guardados consecutivamente en la memoria y se pueden referenciar a través de un índice. El vector es una estructura homogénea, estática, de acceso directo y lineal. (En vez de decir que es de acceso directo, también vale decir que es una estructura “indexada”, ya que para acceder a cada elemento de la estructura se debe utilizar una variable índice que es de tipo ordinal).

El tipo de dato **PUNTERO**, es un tipo de variable usada para almacenar una dirección en memoria dinámica. En esa dirección de memoria se encuentra el valor real que almacena. El valor puede ser cualquiera de los tipos vistos (char, boolean, integer, real, string, record, array o incluso otro puntero). Un puntero es un tipo de dato simple definido por el lenguaje (es decir que no es una estructura de datos).

---

[2] En mi opinión (y la cátedra parece acordar con esto), no tiene mucho sentido clasificarlo dentro de los dicotomías secuencial/directo o lineal/no lineal, ya que si bien tiene campos, el registro es una unidad, y no algo que almacene muchas cosas al estilo de los vectores o las listas.

Sin embargo, a pesar de su aparente simpleza, los punteros pueden causar mucha confusión. Por eso, conviene aclarar un par de cosas.

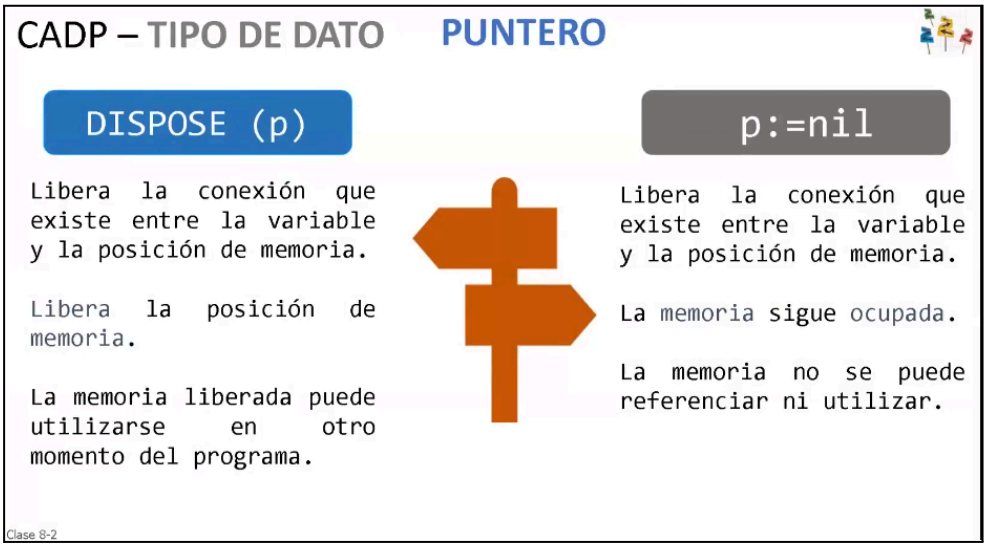
Una variable de tipo puntero ocupa una cantidad de memoria fija (predeterminada), independientemente del tipo de dato al que apunta. Normalmente, en Pascal, un puntero ocupa 4 bytes de MEMORIA ESTÁTICA.

Ahora bien, una variable de tipo puntero puede reservar y liberar MEMORIA DINÁMICA durante la ejecución de un programa para almacenar su contenido, y ahí el tamaño puede variar completamente, dependiendo del contenido que deba almacenar. Obviamente no se reservará la misma cantidad de memoria dinámica para almacenar un integer que para un vector de mil reales, por ejemplo.

Creación de una variable puntero. Crear un puntero significa reservar una dirección de memoria dinámica libre para poder asignarle contenidos a la dirección que contiene la variable de tipo puntero. Para ello, hacemos un *new()*.

Destrucción de una variable puntero. Al eliminarlo, lo que hacemos es liberar esa memoria dinámica que habíamos reservado, es decir, liberamos la memoria dinámica que contenía la variable de tipo puntero. Para ello, contamos con el *dispose()*.

Liberación. Lamentablemente, a la cátedra se le ocurrió ponerle el nombre "liberación" a cuando le asignamos *nil* a un puntero. Esta operación implica cortar el enlace que existe entre el puntero y la memoria dinámica. El espacio de memoria dinámica queda ocupado, pero ya no se puede acceder.

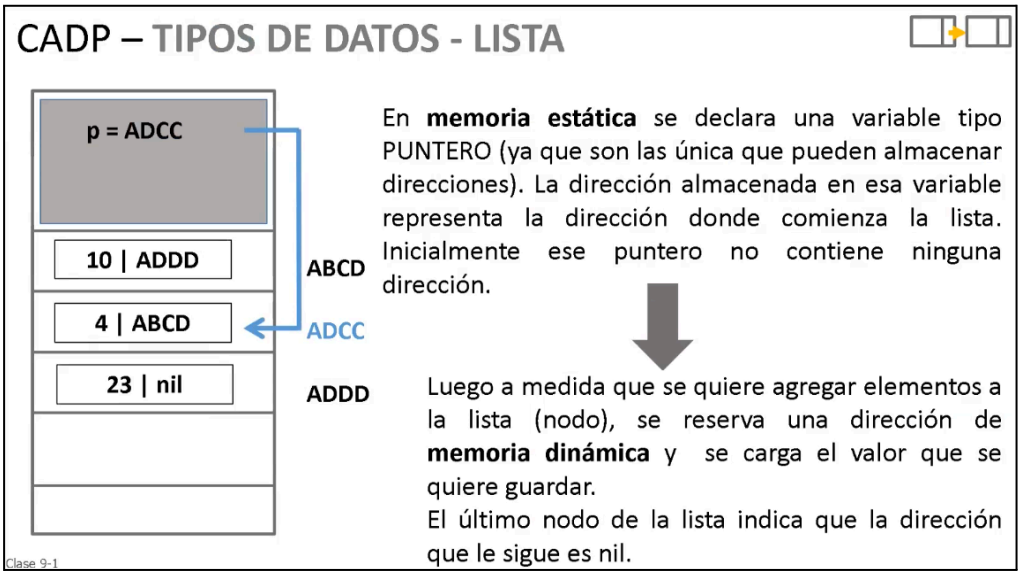


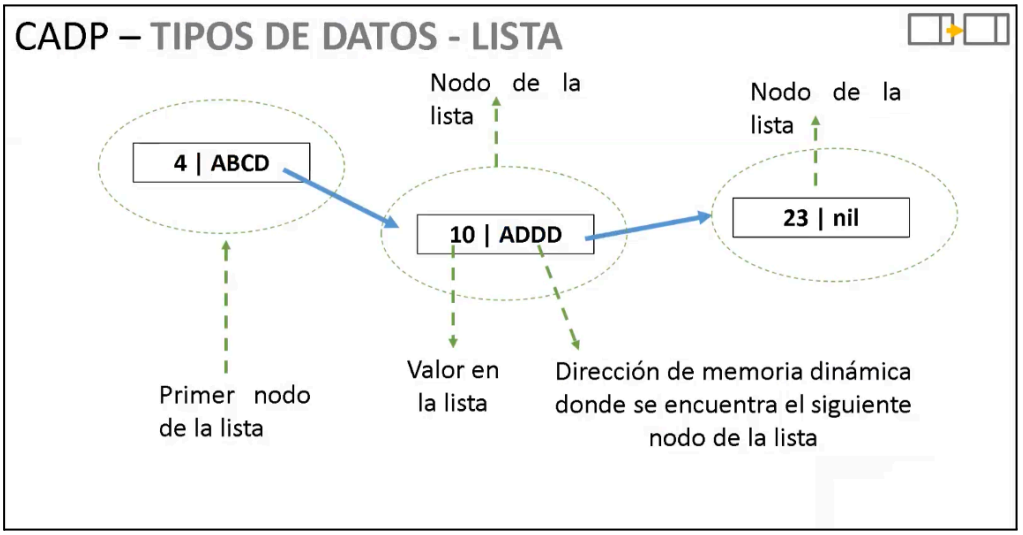
Naturalmente, en esta explicación se asume que el puntero *p* ya apuntaba a algún espacio en memoria, es decir, que primero hacemos new(p) y después p := nil. Pero si lo primero que hacemos con el puntero es p := nil, no causamos ningún problema en la memoria dinámica. Y de hecho así es como lo vamos a usar más adelante con las listas.

Una **LISTA** es una colección de nodos. Cada nodo contiene un elemento (que es el 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.

Toda lista tiene un nodo inicial.

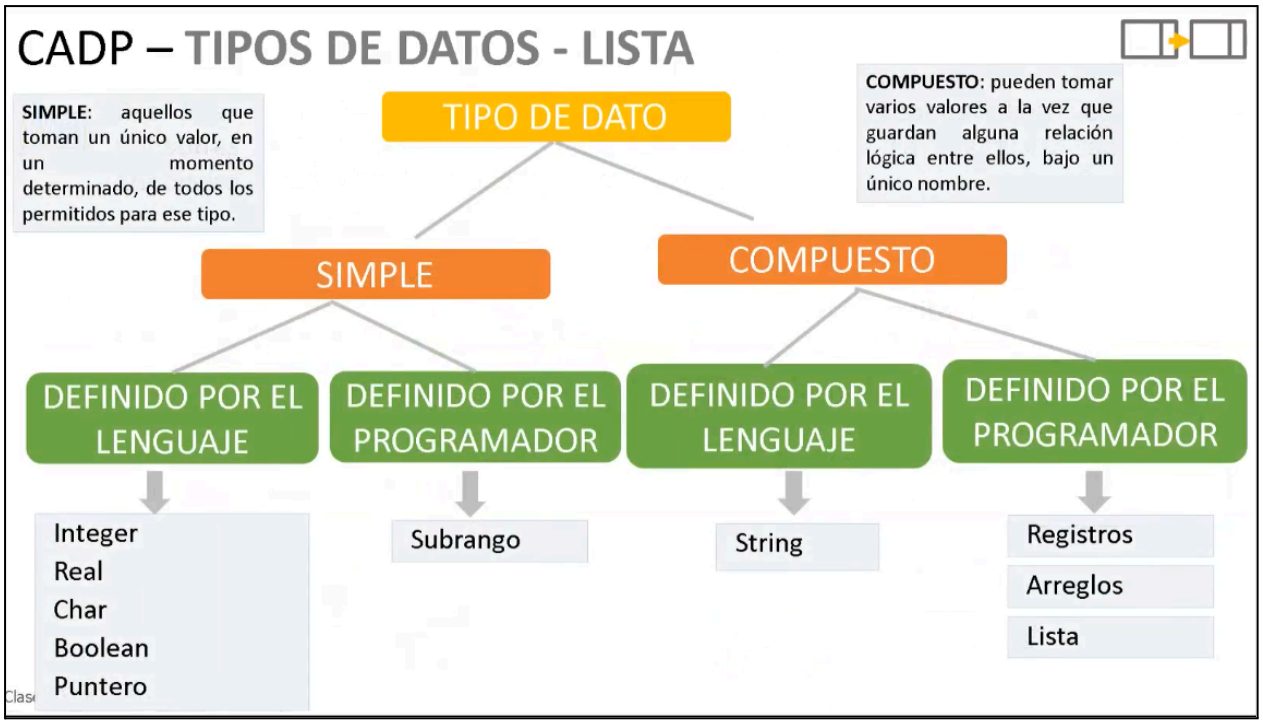
Ahora bien, a diferencia de los vectores, los nodos que componen una lista pueden no ocupar posiciones contiguas de memoria (dinámica). Es decir, pueden aparecer dispersos en la memoria (dinámica), pero mantienen un orden lógico interno.





Por último, ¿cómo las clasificamos? Las listas con estructuras homogéneas (ya que todos los elementos que almacena deben ser del mismo tipo), dinámica, lineal y secuencial.

Ahora sí, con todos los tipos de datos que vimos, ya podemos completar el cuadrito:



Corrección de programas

Cuando se desarrollan los algoritmos hay dos conceptos importantes que se deben tener en cuenta: correción y eficiencia del programa.

Un programa es **correcto** si se realiza de acuerdo a sus especificaciones. O sea, si hace lo que tiene que hacer.

Para poder evaluar si un algoritmo es correcto, existen un conjunto de técnicas. Aquí veremos solamente cuatro:

1) Testing
2) Debugging
3) Walkthroughs
4) Verificación

Se pueden aplicar todas las técnicas (la cantidad de veces que sea necesario) a un mismo algoritmo hasta verificar que este es correcto. No es que solo podemos usar una o dos. Podemos usar todas las que hagan falta hasta asegurarnos de que el algoritmo funciona correctamente.

Vayamos una por una.

El propósito del **TESTING** es proveer evidencias convincentes de que el programa hace el trabajo esperado. Entonces, dado un algoritmo escrito, o un programa, uno analiza los casos límite de ese algoritmo y se fija si funciona bien para todos esos casos.

En otras palabras, lo que se hace es diseñar un plan de pruebas, donde se decide cuáles aspectos del programa deben ser testeados y se tienen que encontrar datos o situaciones de prueba para cada uno de esos aspectos. Asimismo, se debe determinar de antemano el resultado que se espera que el programa produzca para cada caso de prueba.

Cabe resaltar que lo ideal sería diseñar casos de prueba sobre la base de lo que hace (o debería hacer) el programa y no sobre lo que ya se escribió del programa. Por eso lo mejor es hacerlo antes de escribir el programa.

Una vez que el programa ha sido implementado y se tiene el plan de pruebas, entramos en el siguiente bucle:

- se analiza el programa con los casos de prueba
- si hay errores se corrigen

y seguimos repitiendo ese proceso hasta que no haya errores, o sea, hasta corroborar que el algoritmo funciona para todos esos casos límites que habíamos pensado.

El **DEBUGGING** es el proceso de descubrir y reparar la causa del error. Típicamente, lo que se hace es agregar sentencias adicionales en el programa (de tipo *write()* o similar), que permiten monitorear el comportamiento más cercanamente en cada una de las etapas.

Los errores encontrados pueden ser de tres tipos:

- Sintácticos: que se detectan en el momento de la compilación (esto lo avisa el propio programa o compilador).
- Lógicos: generalmente se detectan en la ejecución; una vez que el programa compila y lo ejecutamos, recién ahí cuando lo probamos y no obtenemos el resultado correcto es cuando nos percatamos de estos errores.
- Del sistema: como cuando se corta la luz o algo así; son errores muy raros.

El debugging nos sirve más que nada para los errores lógicos. La técnica nos ayuda a localizar el origen del error.

El **WALKTHROUGH** es el proceso de recorrer un programa frente a una audiencia.

¿Por qué es my utilizada esta técnica? Porque cuando cualquier programador desarrolla su programa, generalmente cree que su programa es correcto y por lo tanto no puede encontrar el error. En cambio, cuando le presenta el programa a otra persona o audiencia (calificada),

esa persona o audiencia no comparte preconceptos y está predispuesta a descubrir errores u omisiones.

A menudo, cuando no se puede detectar un error el programador trata de demostrar que no existe, pero mientras lo hace puede detectar el error, o bien puede que otro lo encuentre.

La **VERIFICACIÓN** es el proceso de controlar que se cumplan las pre y post condiciones del algoritmo.

Básicamente es chequear si resuelve lo que tenía que resolver. Por ejemplo, si estamos con R-Info y el programa tenía que juntar todas las flores de la ciudad, lo probamos y chequeamos si la ciudad quedó sin flores. Entonces es verificar esas poscondiciones que deberían darse después de que el algoritmo se ejecuta.

Eficiencia de programas

Una vez que se obtiene un algoritmo y se verifica que es correcto, recién ahí vamos a analizar la eficiencia del mismo. Pero *antes* de determinar que el algoritmo es correcto, es imposible hablar de la eficiencia de un programa.

Acá está la definición clásica de "eficiencia", tomada del libro de Armando E. De Giusti titulado *Algoritmos, datos y programas*:

8.1 El concepto de eficiencia

Hasta ahora se han desarrollado algoritmos para expresar la solución de problemas simples, sin analizar profundamente cuán "buena" es la solución propuesta, o si existen alternativas "mejores".

Pensar en la optimización de un algoritmo en algún sentido requiere analizar previamente su eficiencia, es decir, la utilización que se hace desde el algoritmo de los recursos del sistema físico donde se ejecuta (básicamente, tiempo de ejecución en máquina requerido y cantidad de memoria utilizada).

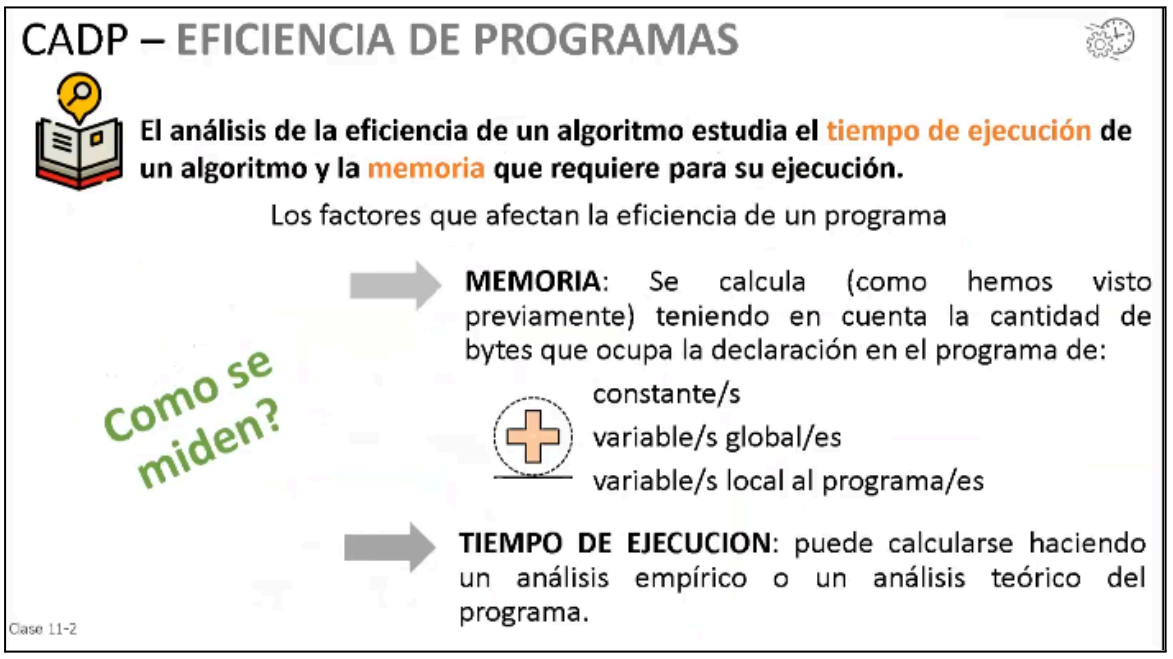
Si bien es cierto que al hablar de eficiencia el mayor énfasis está puesto en el tiempo de ejecución del algoritmo, en un sentido amplio, eficiencia se refiere a la forma de administración de todos los recursos disponibles en el sistema, de los cuales el tiempo de procesamiento es uno de ellos.

Definición

Un algoritmo es eficiente si realiza una administración correcta de los recursos del sistema en el cual se ejecuta.

Hay varios factores involucrados en la eficiencia de un algoritmo. Entre otros, los siguientes:

¿Qué importa más, el tiempo o la memoria? Depende del contexto. En el mejor de los mundos posibles, podremos conseguir un algoritmo que sea más rápido y liviano que los demás, pero eso no suele ser así: normalmente tenemos que elegir entre uno más lento pero liviano, y uno rápido pero pesado. El contexto y los recursos que tengamos a mano nos indicarán cuál elegir.



¡Hacé memoria!

A la hora de saber cuánto ocupa un programa en memoria, puede que nos interese saber cuánto ocupa en memoria estática y cuánto en memoria dinámica. Cabe aclarar que no estamos hablando de dos memorias distintas. La *memoria estática* es el espacio fijo que el programa ocupa en memoria y se define al momento de compilarlo (es decir que no varía durante la ejecución del programa), mientras que la *memoria dinámica* es el espacio que el programa va ocupando en memoria durante la ejecución del programa (y por ello es que puede variar durante la ejecución del programa, porque puede reservar o liberar espacio según lo que vaya haciendo).

Para el cálculo de MEMORIA ESTÁTICA solamente se evalúan las declaraciones de las constantes, las variables globales o las variables del programa principal[3].

Para el cálculo de MEMORIA DINÁMICA, se evalúan únicamente las operaciones de *new()* y *dispose()* dentro del programa.

¿Cómo sabemos cuánto pesa cada cosa? Por lo general, nos dan una tabla como la siguiente:

Referencia	
Char	1 byte
Integer	4 bytes
Real	8 bytes
Boolean	1 byte
String	Longitud + 1
Puntero	4 bytes

Si no se especifica la longitud de la string, esta ocupa 256 bytes, ya que el máximo de caracteres es 255

Y con eso vamos haciendo las cuentas. Por ejemplo:

```
var
      num1, num2 : integer; // 4, 4
      result : real; // 8
      es_verde : boolean; // 1
```

Todo esto suma 4 + 4 + 8 + 1 bytes, o sea: 17 bytes de memoria estática.

[3] ¿Esto es realmente así? ¿Las instrucciones no ocupan nada? No, en realidad es más complejo. Para ser rigurosos, deberíamos hacer un análisis parecido al que hacemos en Organización. Pero no importa, así es la simplificación que vemos en CADP. (Menos mal).

Casos especiales: registro, vector y un puntero a nodo.

Un **registro** se calcula sumando sus campos, así:

```
type
      persona = record // este registro pesa 8 bytes
            dni : integer; // 4
            edad: integer; // 4
      end;

var
      p1, p2 : persona; // 8, 8
```

Lo que está en `type` no suma, pero debemos calcularlo para saber cuánto sumar cuando se declaren variables de ese tipo. En este caso, el registro persona tiene 2 enteros, por eso el registro ocupa 8 bytes; y en las variables del programa principal hay dos variables de tipo persona, con lo cual este programa ocupa 16 bytes de memoria estática.

Para el **vector**, lo mismo: sumamos todo lo que tiene adentro. O sea, multiplicamos la cantidad de elementos que tiene por lo que pesa cada elemento. Podemos dividir este proceso en tres pasos.

Paso 1: saber cuántos elementos tiene. Esto puede hacerse con la misma técnica que utilizamos cuando calculamos la N para una estructura de control como el for. O sea, si el vector es así:

```
vec1 = array [1..10] of real;
```

Es obvio que tiene 10 elementos.

Ahora bien, si es medio raro como este

```
vec2 = array [10..17] of ^persona;
```

conviene llevarlo a la "forma estándar", razonando así: quiero que el primer elemento empiece en 1. Sabemos que 10 - 9 = 1, entonces hay que restarle 9 a ambos "índices" y obtenemos que 17 - 9 = 8. Conclusión: vec2 tiene 8 elementos.

Paso 2: saber cuánto pesa cada elemento. En el primer caso es fácil, lo dice la tabla: cada elemento pesa 8 bytes. En el segundo, aunque parezca raro, también es facil: es un array de punteros, con lo cual lo obtenemos de la tabla: cada elemento pesa 4 bytes.

Paso 3: multiplicar. El vec1 ocupa $10\ elementos \cdot 8\ bytes = 80\ bytes$, mientras que el vec2 ocupa $8\ elementos \cdot 4\ bytes = 32\ bytes$.

El **nodo** es aquello con lo cual se forman las listas, y es, a fin de cuentas, un registro, por eso su peso es igual a la suma de sus campos. Ahora bien, a diferencia del resto de los registros, los nodos siempre contarán como memoria dinámica, ya que "nacen" con la operación *new()* y "mueren" con *dispose()*. Esto se debe a que nunca declaramos una variable de tipo nodo directamente, sino que creamos una variable de tipo lista, que es un puntero a un nodo.

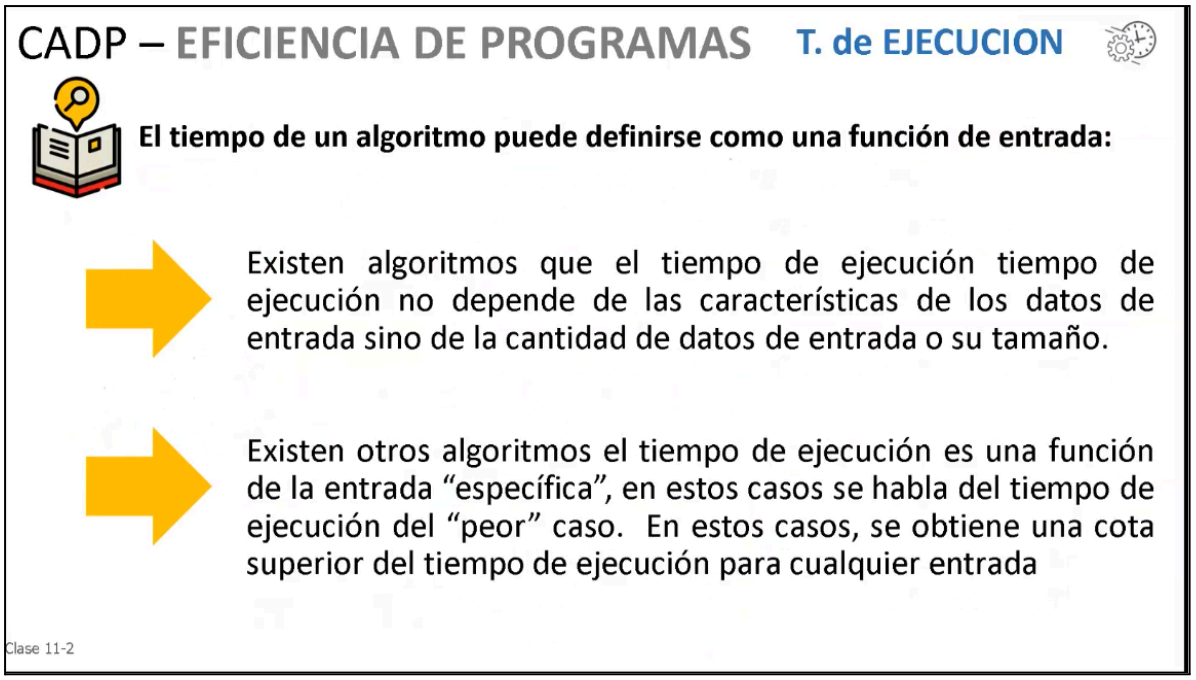
Supongamos que tenemos el siguiente programa:

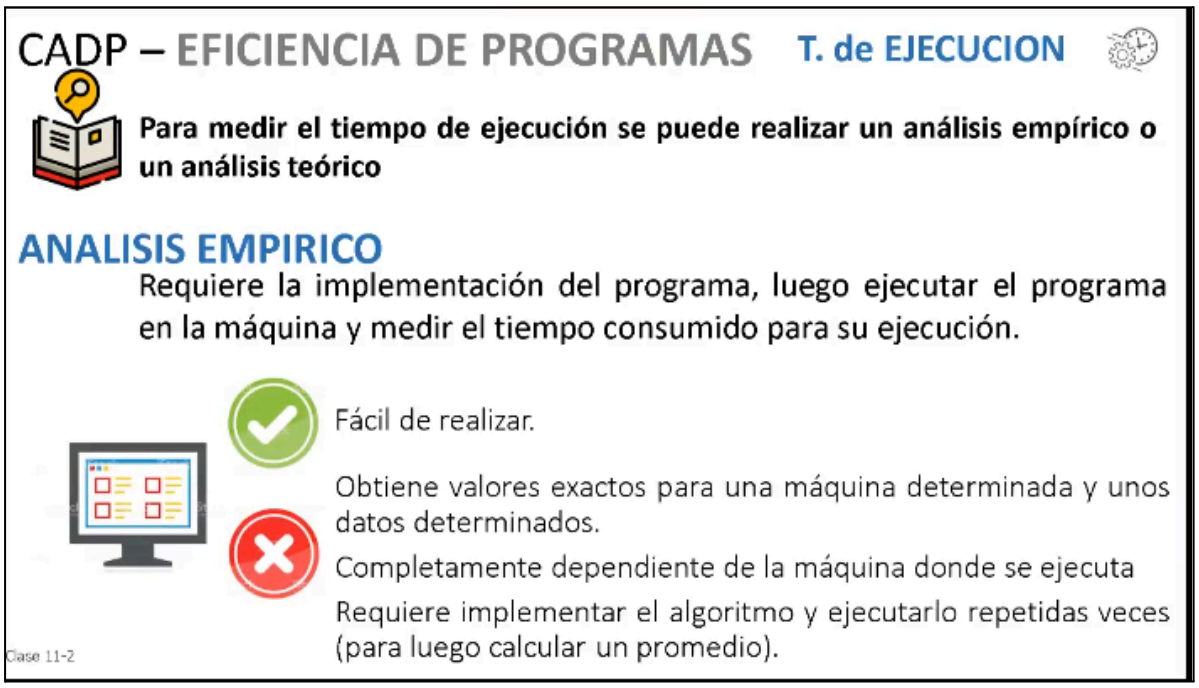
```
program ejemplo_del_nodo;

type
      lista = ^nodo; // 4

      nodo = record // 12
            elem : real; // 8
            sig : lista; // 4
      end;

var
      L, nuevo, viejo, anterior : lista; // 4, 4, 4, 4

begin
      // Memoria dinámica inicial: 0
      nuevo := nil;
      new(nuevo); // + 12
      nuevo^.elem := 2;
      dispose(nuevo); // - 12
      // Memoria dinámica final: 0
end.
```

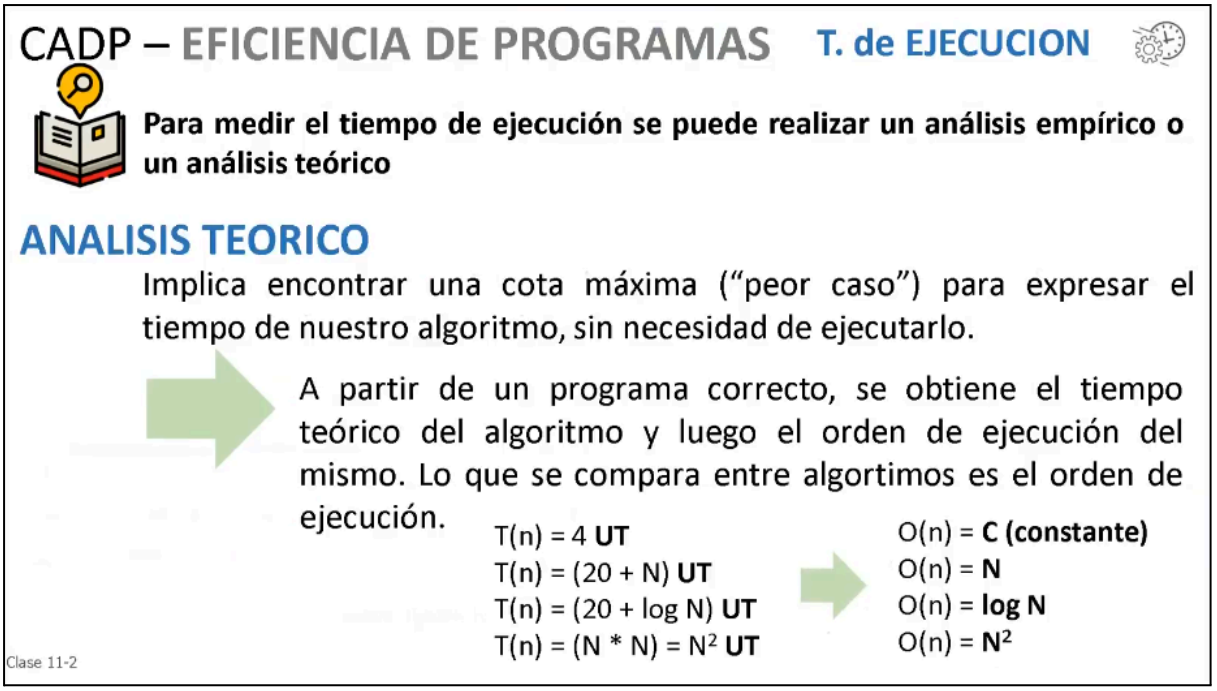
Aquí podemos ver que el programa ocupa 16 bytes de memoria estática.

Con respecto a la memoria dinámica, empieza en 0 (como siempre), luego se reservan 12 bytes para el nodo "nuevo", después se libera ese espacio, así que vuelve a quedar en 0.

Ahora nos concentraremos en el **TIEMPO DE EJECUCIÓN**.







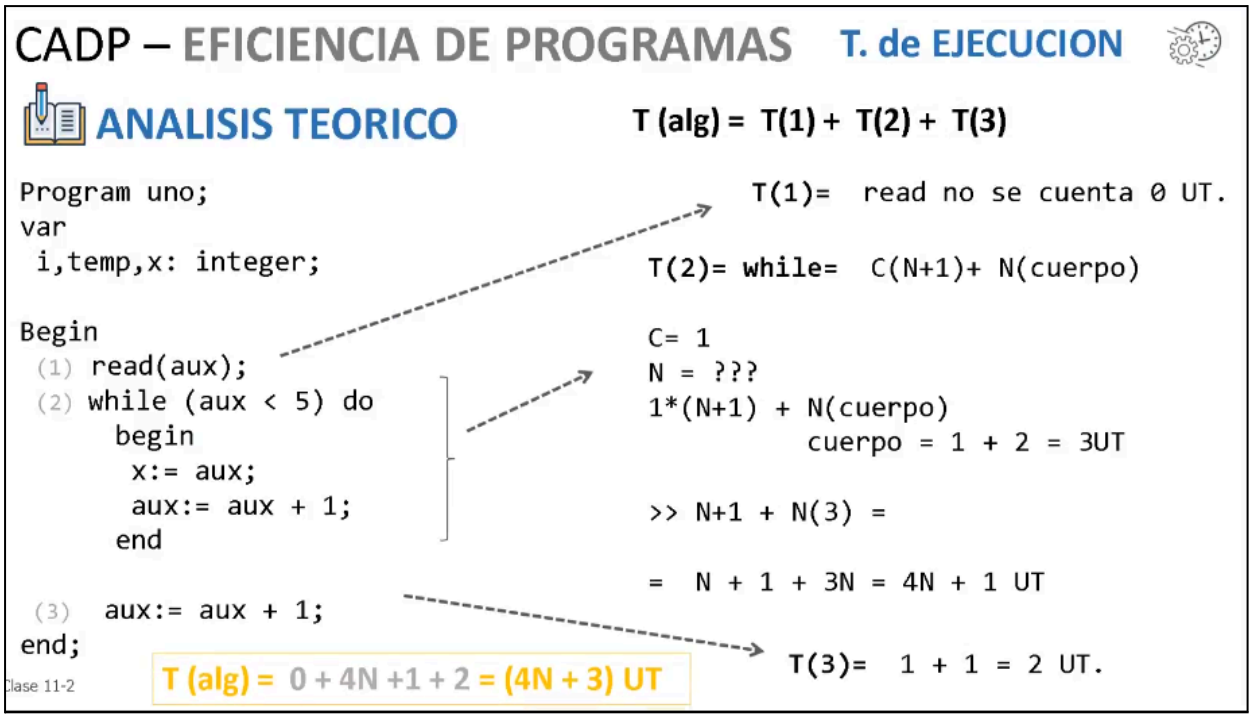
PARA CALCULAR EL TIEMPO en unidades de tiempo (UT), usamos la siguiente tabla de tiempos de ejecución:

- Tanto la asignación de valor a una variable, como una operación aritmético-lógica cualquiera (sea suma, resta, conjunción, disyunción etc.), demandan 1UT. Write(), read(), new() y dispose() no cuentan.
- Tiempo del if ... then ... = C + tiempo del cuerpo.
- Tiempo del if ... then ... else ... = C + max(then, else).
- Tiempo del for = (3N + 2) + N(cuerpo del for).
- Tiempo del while = C(N+1) + N(cuerpo del while).
- Tiempo del repeat ... until = C(N) + N(cuerpo del repeat).

Nótese que

- C es el tiempo de evaluar la condición.
- N es la cantidad de veces que se ejecuta algo (sea el bloque del for, del while, etc.).
- max(a, b) elige el número máximo entre a y b.

¿Y qué pasa si N no queda con un valor determinado? No pasa nada, lo dejamos así, anotado con la N, como en el siguiente ejemplo:



Cuando pasa esto, se dice que el algoritmo tiene un tiempo lineal.

Si podemos dejarlo expresados simplemente con un número, como en los casos sencillos, decimos que tiene un tiempo constante.

ANEXO: CALCULAME N NÉSTOR

Ponele que hay que calcular el N de **un *for* común que va de 1 a n (donde n es un entero positivo**, y N representa la cantidad de veces que se ejecuta el cuerpo del *for*).

Obviamente será n veces, ya que se ejecuta cuando i vale 1, cuando vale 2, y así siguiendo hasta llegar a n; y también se ejecuta cuando i vale n. O sea que nos queda una correspondencia uno a uno entre el índice y la cantidad de ejecuciones.

Gráficamente, queda una cosa así:

						Total
Valores de válidos de i	1	2	3	…	n	n valores válidos
¿S ejecuta el cuerpo del *for*?	Sí	Sí	Sí	…	Sí	n ejecuciones

Eso es lo que podemos observar y corroborar en el siguiente programa:

```
program calculando_el_for_común;

var i, N : integer;

begin
   N := 0;
   for i := 1 to 15 do
       N := N + 1;
   writeln('El cuerpo del for se ejecutó ', N, ' veces');
end.
```

Que imprime:

```
El cuerpo del for se ejecutó 15 veces
```

Ahora compliquemos las cosas un poco más.

Supongamos que vamos de 13 a 99. ¿Qué hacemos? Tenemos dos opciones: llevarlo a la forma estándar o aplicar la fórmula de la cátedra.

Opción 1: para llevarlo a la forma estándar, aplicamos la misma operación a ambos miembros. Primero tenemos que pensar cómo llevar 13 a 1. Obviamente, podemos hacer esto mentalmente: sabemos que hay que restarle 12. Entonces corroboramos: 13 - 12 = 1. Y ahora le aplicamos la misma operación a la otra parte: 99 - 12 = 87. Y así obtenemos un *for* común que va de 1 a 87, y listo, sabemos que la respuesta es 87. ¿Lo probamos?

```
program calculando_el_for_loco;

var i, N : integer;

begin
   N := 0;
   for i := 13 to 99 do
       N := N + 1;
   writeln('El cuerpo del for se ejecutó ', N, ' veces');
end.
```

Y nos dice:

```
El cuerpo del for se ejecutó 87 veces
```

Opción 2: usar la fórmula: *límite superior* - *límite inferior* + *1*. Y funciona: 99 - 13 + 1 = 87.

Lo que no me gusta de esta opción es que te hace memorizar algo que uno fácilmente puede obtener si lo piensa dos segundos. ¿Y para qué lo quiero obtener pensando? Para poder calcular también el N de las otras estructuras, o de casos más difíciles.

Veamos un último *for*. Ponele que este for es así:

```
program calculando_el_for_re_loco_mal;

var i, N : integer;

begin
   N := 0;
   for i := 77 downto -19 do
       N := N + 1;
   writeln('El cuerpo del for se ejecutó ', N, ' veces');
end.
```

Apa, te quiero ver *límite superior* - *límite inferior* + *1*: $(-\ 19)\ -\ (77)\ +\ 1\ =\ \ -\ 95$

¿Se ejecuta $-\ 95$ veces? No. ¿Acaso serán 95 veces? Tampoco.

¿Será que había que hacer al revés? $77\ -\ (-\ 19)\ +\ 1\ =\ 97$. Sí, ahí funciona.

De hecho, nuestro programa imprime:

```
El cuerpo del for se ejecutó 97 veces
```

¿Cómo sería con la otra técnica? Bueno, hay que pensar un poco, y modificar algunas cosas. Si llevamos el primer número a 1, nos queda así: 77 - 76 = 1 y -19 -76 = -95, que tampoco es. Lo que sucede es que si hacemos esto vamos a estar yendo de 1 a un n negativo. Y la propuesta que habíamos mencionado sólo vale para n positivo. Entonces, ¿deja de servir? No, simplemente tenemos que intercambiar los lugares.

Si el *for* original plantea esto:

```
for i := 77 downto -19 do
```

nosotros lo vamos a llevar a esto:

```
for i := n donwto 1 do // donde n es un entero positivo
```

Y listo. Entonces hacemos la cuenta para llevar -19 a 1 y obtenemos: -19 + 20 = 1, y 77 + 20 = 97, que es a donde queríamos llegar.

Como regla general: calculá esto de todas las maneras que se te ocurran. Si siempre da lo mismo, entonces debe estar bien. Si hay discrepancia en los resultados, puede estar mal.

Ahora sí, pasemos a lo hardcore: *while and repeat*.

¿Cómo hacemos acá? Mi recomendación es llevarlo al formato estándar, donde *i* empieza en 1 y termina en n, siendo n un entero positivo.

Primero que nada, tenemos que ver cuáles serán los valores válidos del índice, o sea, el **primer valor válido** y el **último valor válido**. En el *for* vienen indicados de forma bastante explícita. En el *while* puede que no.

Miremos el siguiente programa:

```
program calculando_el_while;

var i, N : integer;

begin
   N := 0;
   i := 0;
   while (i < 7) do
   begin
       N := N + 1;
       i := i + 1;
   end;
   writeln('El cuerpo del while se ejecutó ', N, ' veces');
end.
```

Quizás no nos parezca tan difícil porque podemos calcularlo con los dedos y hacer algo tipo: con 0 se ejecuta, con 1 también, con 2, con 3, etc., y así hasta darnos cuenta que:

```
El cuerpo del while se ejecutó 7 veces
```

Pero si tenemos números más grandes, no da andar contando. Lo que sí podemos hacer es buscar el primer valor válido (que es 0) y el último valor válido (que es 6). Esto es como tener un *for* de 0 a 6. ¿Y cómo lo llevamos a la forma estándar? Sumamos 1 a ambos miembros y obtenemos que esta cosa va de 1 a 7. O sea que se ejecuta 7 veces. O sea, N = 7.

Lo mismo con el repeat until. Veamos el siguiente ejemplo:

```
program calculando_el_repeat;

var i, N : integer;

begin
   N := 0;
   i := 33;
   repeat
       N := N + 1;
       i := i + 1;
   until (i > 172);
   writeln('El cuerpo del repeat se ejecutó ', N, ' veces');
end.
```

Pensemos: **primer valor válido** = 33, **último valor válido** = 172

Estandarizamos: 33 - 32 = 1 172 - 32 = 140

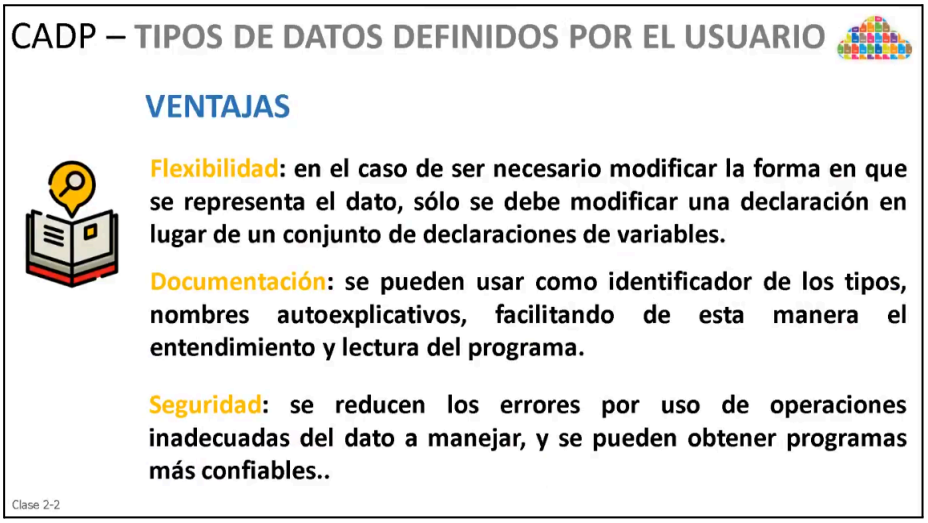
¿Y qué dice nuestro amigo Pascalito?

```
El cuerpo del repeat se ejecutó 140 veces
```

Todo en orden.

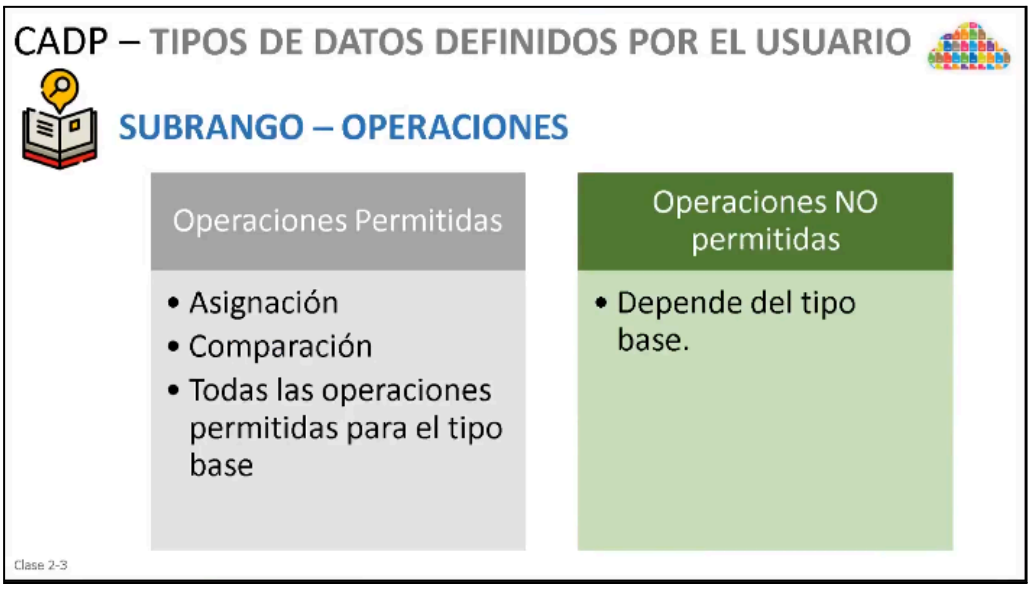
ANEXO 2: + INFO

Tipos de datos definidos por el usuario



Para poder hacer un subrango, necesito que el tipo usado como base sea un tipo de dato simple y ordinal.

Las operaciones permitidas dependen del tipo base.



Supuestamente una de las ventajas de usar subrango es que si el usuario ingresa un dato fuera del subrango, el programa genera una excepción, y nosotros podríamos decirle al manejador de excepciones qué es lo que queremos que pase: si queremos que el programa crashee, o que lea de nuevo los datos, lo que fuera. Pero de todas formas, la cátedra no pide que sepamos usar el manejo de excepciones, aunque sí quiere que usemos subrango cuando sea oportuno.

CLAUDE SOBRE ESTE TEMA

En Pascal, el manejo de excepciones para subrangos funciona de manera específica y depende del compilador que uses. Te explico cómo funciona:

Comportamiento con subrangos

Cuando defines un subrango como:

```
type
  TEdad = 0..120;
  TMes = 1..12;
```

Y intentas asignar un valor fuera del rango, el comportamiento depende de las directivas del compilador:

Directivas de compilación relevantes

{$R+} (Range Checking ON) - Activa la verificación de rangos
{$R-} (Range Checking OFF) - Desactiva la verificación de rangos

Con `{$R+}` activo:
- Se genera una excepción `ERangeError` cuando se viola el rango
- El programa se detiene con un error de runtime si no se maneja
- Puedes capturar esta excepción con bloques try-except

Opciones de manejo

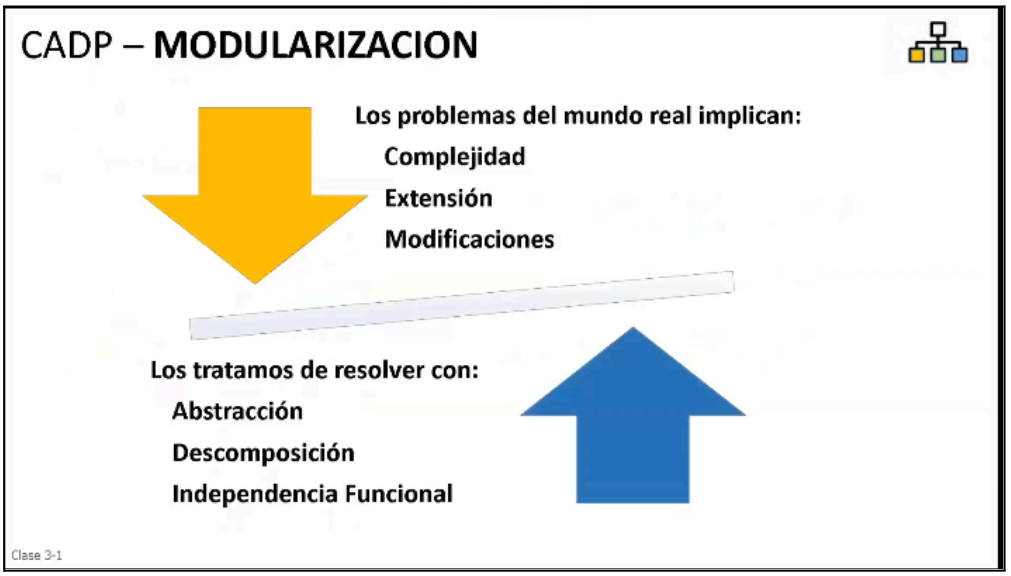
Cuando capturas la excepción, puedes:

1. **Terminar el programa** graciosamente
2. **Pedir el dato nuevamente** en un bucle
3. **Asignar un valor por defecto**
4. **Registrar el error** y continuar

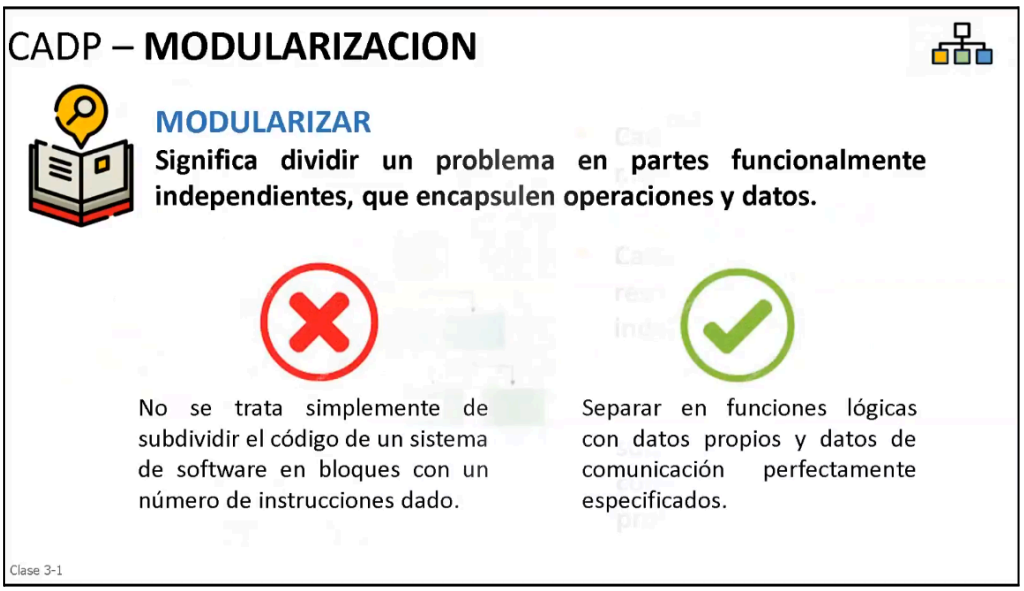
El manejo de excepciones para subrangos es una característica útil de Pascal que te permite crear programas más robustos, pero debes activar explícitamente la verificación de rangos para que funcione.

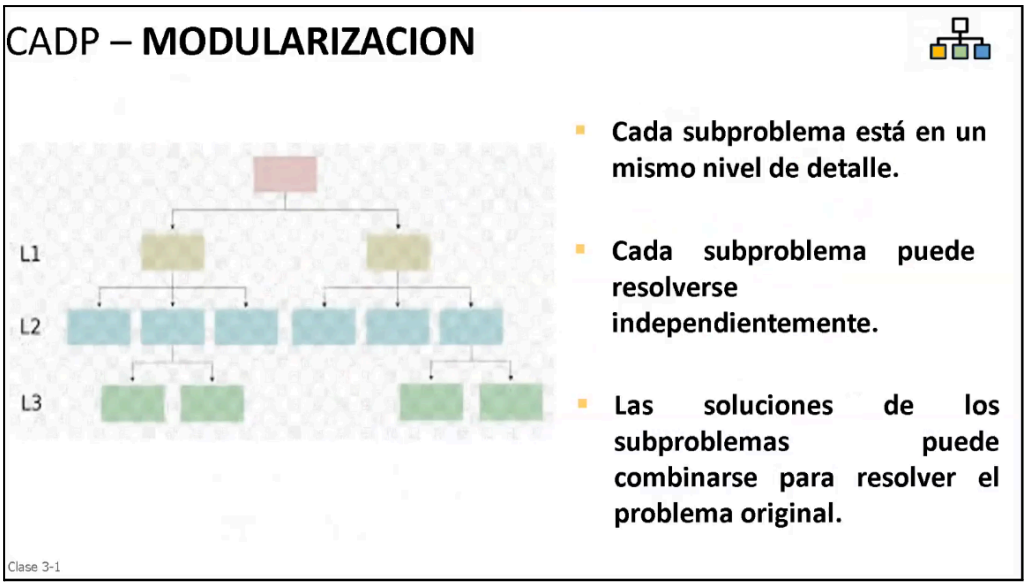
¿Por qué modularizamos?

Porque la realidad es compleja, y tratamos de simplificarla mediante abstracciones y de dividirla en partes para poder manejarla mejor. Usamos la metodología top-down.



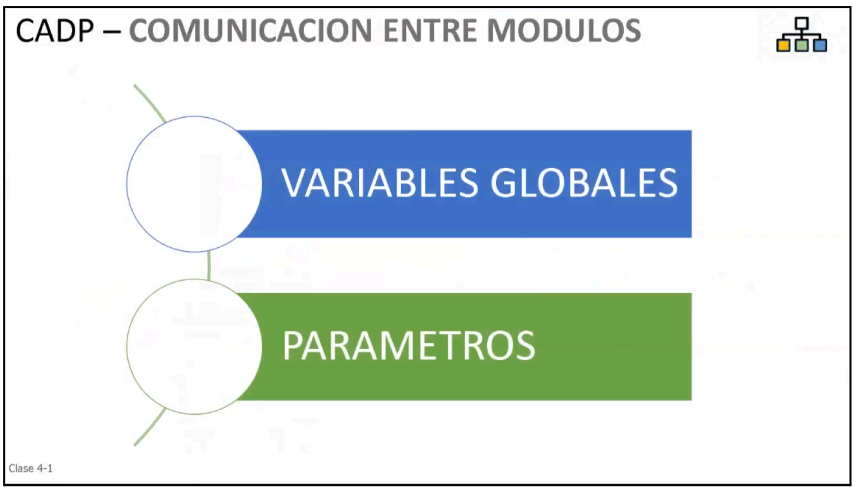
Modularizar no es dividir un programa en módulos de 10 lineas cada uno. Cada módulo puede tener su propia extensión y complejidad. Lo que sí tratamos de hacer es que cada uno tenga un sentido y una funcionalidad, y que sean lo más independiente que podamos.





Comunicación entre módulos

Hay dos alternativas principales. Una es usando variables globales, la otra es usando parámetros. Pero como ya vimos, las variables globales pueden ser muy problemáticas, porque todos los módulos tienen acceso a esas variables y pueden modificarla, con lo cual es muy probable que a veces se modifiquen cosas sin que nos demos cuenta y el programa empiece a tener errores.
Entonces, las variables globales, si bien existen, debido a sus desventajas las vamos a descartar para el uso de la comunicación, y cuando querramos dar la comunicación entre módulos, lo haremos mediante la utilización de parámetros.



CADP – COMUNICACION ENTRE MODULOS

VARIABLES GLOBALES

```
Program ejemplo1;
 Var
  x:integer;

 Procedure uno;
  Begin
    x:= x+1;
    write (x);
  End;
 Procedure dos;
  Begin
    x:= x MOD 10;
    write (x);
  End;
 var
  x: integer;

Begin
  x:=9;
  uno;
  write (x);
End.
```

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.

CADP – COMUNICACION ENTRE MODULOS

PARAMETROS

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.

CADP – COMUNICACION ENTRE MODULOS

PARAMETROS

```
Program ejemplo2;

 Procedure uno (PARAMETRO1; PARAMETRO2);
  Begin
    …
  End;
 Procedure dos (PARAMETRO);
  Begin
    …
  End;
 var
  x,y,z: integer;

Begin
  ….
  uno(x,y);
  dos(z);
  …
End.
```

Cada módulo indica que necesita recibir

Cada módulo indica que devuelve

No existe el problema donde un se pueda modificar el valor sin darse cuenta.

No me gusta mucho cómo la profe explicó este tema, así que saqué algunas cosas del libro.

4.4 Parámetros

Generalmente, los módulos, tanto procedimientos como funciones, necesitan para operar algunos datos que están disponibles en el programa o subprograma que lo invoca. En ambos casos es necesario "comunicar" módulos. La forma en que se realiza esta comunicación es a través de parámetros.

Definición

Se denomina parámetros a la serie de datos con los que se comunican los módulos.

Los parámetros de un módulo deben definirse, como se indicó anteriormente, en el encabezado del mismo. Cada parámetro debe especificar el tipo de datos con el que se corresponde. En la invocación del módulo deben definirse también estos parámetros. Los parámetros que se definen en el llamado del módulo reciben el nombre de **parámetros actuales**, en tanto que los parámetros descritos en el encabezado del módulo invocado (procedimiento o función) se denominan **parámetros formales**.

La mayoría de los lenguajes de programación exigen que el número de parámetros actuales y formales sea el mismo y de igual tipo. Pascal chequea esta condición que, en caso de no cumplirse, impide ejecutar el programa. En otros lenguajes, si el número de parámetros actuales es menor a la cantidad de parámetros formales, algunos de estos quedan con valores nulos.

4.4.2.1 Pasaje de parámetros por valor

Cuando un parámetro es pasado por valor, el valor del parámetro actual es (en el módulo que llama) utilizado para inicializar el parámetro formal correspondiente (en el módulo llamado), que luego actúa como una constante local en el subprograma. Este parámetro se denomina de entrada. La Figura 4.4 a) presenta un gráfico de la semántica de este pasaje.

Atención: convendría decir que actúa como una *variable* local en el subprograma, ya que puede modificarse

CADP – MODULARIZACION COMUNICACION

```
Program porValor;

procedure uno (num: integer);
Begin
    if (num = 7)  then
      num:= num + 1;
    write (num);
end;
var
   x: integer;
begin
  x:= 7;
  uno (x);
end.
```

Procedimiento uno
Variables locales
Parámetros

num = 8

Programa ppal
Variables globales
Variables de prog

x = 7

MEMORIA

Clase 4-2

4.4.2.4 Pasaje de parámetros por referencia

Es una segunda implementación del pasaje de entrada-salida. En lugar de transmitir el valor del dato, el parámetro actual transfiere la dirección real de memoria donde la variable se encuentra. De esta forma, el parámetro actual y formal "comparten" la misma zona de memoria. Por consiguiente, cualquier modificación que se realiza en el proceso es "vista" por el modulo llamador.

Cuando se transmite una variable a un procedimiento como parámetro por referencia, los cambios que se efectúan en dicha variable, dentro del procedimiento, se mantienen incluso después que este haya terminado. Es decir, los cambios "afectan" (modifica el valor de la variable) al módulo que lo invocó. La Figura 4.5 ilustra este pasaje.

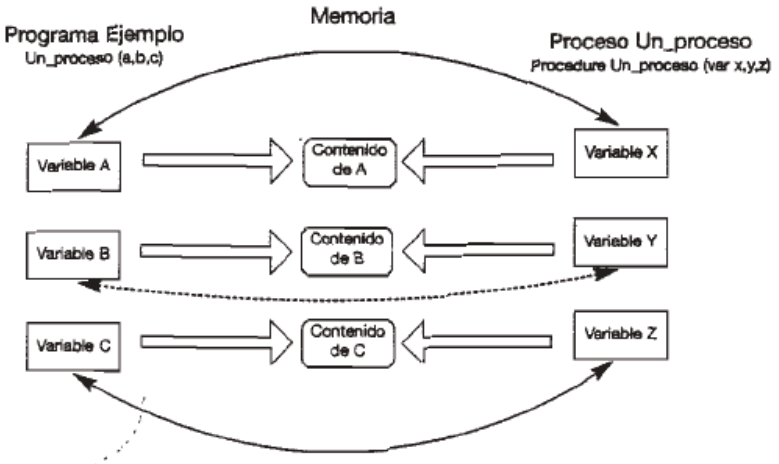


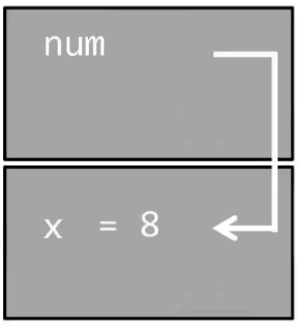
Figura 4.5

CADP – MODULARIZACION COMUNICACION

```
Program porReferencia;

procedure uno (var num: integer);
Begin
    if (num = 7)  then
    num:= num + 1;
    write (num);
end;
var
   x: integer;
begin
  x:= 7;
  uno (x);
end.
```

Procedimiento uno
Variables locales
Parámetros

Programa ppal
Variables globales
Variables de prog



MEMORIA

Y recordemos que, como regla general, el pasaje de parámetros es por posición (no importa el nombre), a cada parámetro actual le corresponde un parámetro formal (por eso deben tener la misma cantidad), y tienen que ser del mismo tipo, por eso:

- 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.

Vectores