

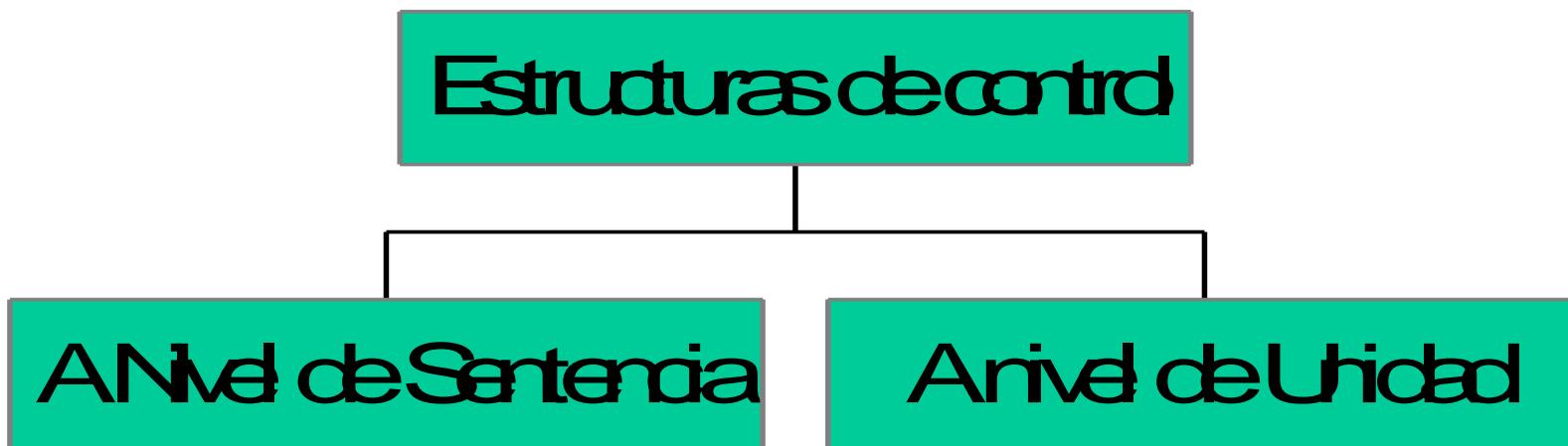
# ESTRUCTURAS DE CONTROL

## CYPLP

# ESTRUCTURAS DE CONTROL

Los lenguajes de programación convencionales permiten estructurar el código en relación al flujo de control entre los diferentes componentes de un programa a través del uso de estructuras de control

Son el medio por el cual los programadores pueden especificar el flujo de ejecución entre los componentes de un programa.



# ESTRUCTURAS DE CONTROL

- **A Nivel de Unidad:**
- Cuando el **flujo** de control se pasa **entre unidades** (**rutinas, funciones, procedimientos** Etc.) también es **necesario estructurar el flujo** entre ellas

Las formas de control son:

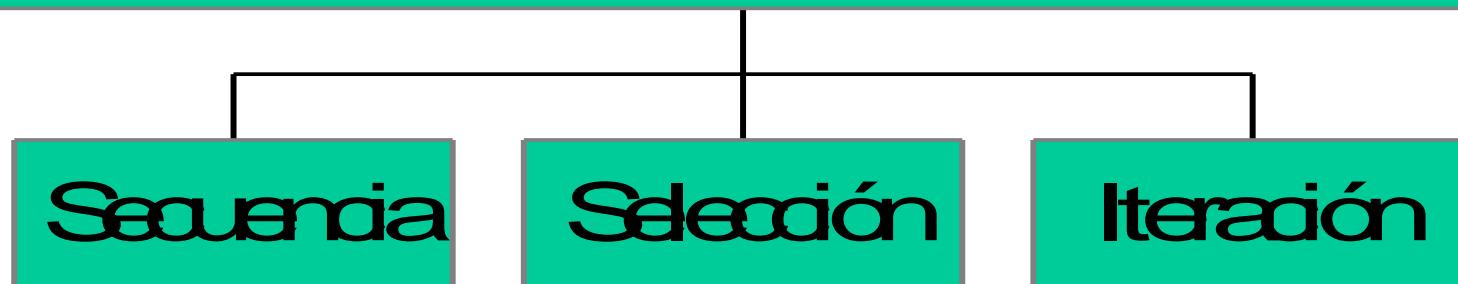
- **Pasajes de Parámetros.**
- **Call-Return**
- **Excepciones**
- **Otros más**

- *Se ve en más detalle en otras clases*

# ESTRUCTURAS DE CONTROL

- A Nivel de Sentencia:
  - Se dividen en tres grupos

Estructuras de control a nivel de sentencia



# ESTRUCTURAS DE CONTROL

- **Secuencia:** estructuras de control que **permiten ejecutar una serie de instrucciones en un orden específico**, de arriba hacia abajo, sin ningún tipo de desviación.
- **Iteración (o bucles):** estructuras de control que **permiten repetir un bloque de código múltiples veces hasta que se cumpla una condición de salida**. Esto permite la ejecución repetida de una serie de instrucciones sin tener que escribir las mismas instrucciones una y otra vez. Ejemplos **for**, **while**, **do-while** y similares.
- **Selección (o decisiones):** estructuras de control que **permiten tomar decisiones basadas en ciertas condiciones**. Estas estructuras dirigen el flujo del programa hacia diferentes caminos según el resultado de la evaluación de las condiciones especificadas. Ejemplos **if-else**, **switch-case** (o **select-case** en algunos lenguajes), y otras estructuras similares

# 1 SECUENCIA

- Es el **flujo de control más simple.**
- Ejecución de **una sentencia a continuación de otra.**
- El **delimitador** más **general** y más usado es el “;”

*Sentencia 1;*

.....

*Sentencia n;*

Hay lenguajes que:

1. **NO usan delimitador**
2. **Por cada línea sólo** debe haber **1 instrucción.** (Se los llaman **orientados a línea**). Ej: Fortran, Basic, Ruby, Phyton
3. **Por cada línea** pude haber **más de 1 instrucción** (ejemplo: ( $p > q$ ) ?  $p^*$  :  $q^* = 0$  ;)
4. **Compuestos:** Bloques de instrucciones

# 1 SECUENCIA – SENTENCIAS COMPUESTAS

## *Sentencias Compuestas*

- no permitido en todos los lenguajes
- Se pueden **agrupar varias sentencias en una** con el uso **delimitadores**.
- Depende de las reglas de cada lenguaje:
  - **Begin y End.** En Ada, Pascal
  - **{ }** en C, C++, Java, etc.

### Ej. S. Compuestas en C

```
{  
pi=3.14;  
c=x+y;  
++i;  
}
```

### Ej. S. Compuestas en Pascal:

```
Begin  
Readln (Numero);  
Numero:= Numero+1;  
Write ('El número es ',Numero)  
End.
```

# SECUENCIA - SENTENCIA DE ASIGNACIÓN

- **Asignación:** Sentencia que **produce cambios** en los **datos de la memoria.** ( $x = a + b$ )

*En general, asigna al l-valor de un objeto dato (x) el r-valor de una expresión. ( $a + b$ )*

- Sintaxis en diferentes lenguajes

<b>A := B</b>	Ej: Pascal, Ada, etc.
<b>A = B</b>	Ej: Fortran, C, Prolog, Python, Ruby, etc.
<b>MOVE B TO A</b>	COBOL
<b>A ← B</b>	APL
<b>(SETQ A B)</b>	LISP

# SECUENCIA - SENTENCIA DE ASIGNACIÓN

## DISTINCIÓN EN ENTRE ASIGNACIÓN Y EXPRESIÓN

- En cualquier **lenguaje convencional** existen diferencias entre **sentencia de asignación** y **expresión**.

Ejemplos:

- $(x + 3)^*2$  // expresión
- $a > b \ \&& c < d$  // expresión
- $y = (x + 3)^*2;$  // sentencia

# SECUENCIA - SENTENCIA DE ASIGNACIÓN

## DISTINCIÓN EN ENTRE SENTENCIA DE ASIGNACIÓN Y EXPRESIÓN

- **Asignación:** devuelve el r-valor de una expresión y lo asigna al l-valor modificando así el valor de esa posición de memoria.
- **Expresión:** devuelve el valor obtenido como resultado de combinar otros valores a través del uso de operadores.
- En otros lenguajes, como en C, se define la sentencia de asignación como una “expresión con efectos colaterales”.

# EJEMPLO EFECTOS COLATERALES EN C

```
1 #include <stdio.h>
2
3 int main()
4 {
5     //ejemplo asignación múltiple
6     /*int a,b,c;
7     a=b=c=0;*/ // linea comentada
8     printf("a es igual a %d, b es igual a %d, c es igual a %d \n",a,b,c );
9     /*
10
11     //ejemplo de asignación que retorna un valor
12     /*int i=0;
13     if(i=30) [ linea comentada
14     printf("Es verdadero porque i toma el valor 30 y al ser mayor a 0 se considera true. \n Valor de i es %d", i);
15     }*/
16
```

## Ejemplo de asignación múltiple:

- **0** se devuelve como r-valor a todos, primero se asigna a C, luego C asigna a B, luego B asigna a A y **todos valen 0**
- En C el operador de asignación asocia de derecha a izquierda

Para ejecutar sacar comentarios /\*  
Sino el compilador de C generará un error  
*undeclared*

## Ejemplo asignación que retorna valor:

- **no es comparación**, C usa == para comparar
- Primero asigna 30 a i, luego interpreta que **I es distinto de 0**, ser mayor que 0 da **verdadero**,
- **imprime 30**

Continua el código.....

# EJEMPLO EFECTOS COLATERALES EN C

```
17 //ejemplo de lvalue en el lado izquierdo  
18 /*int p=0; //Declaración de variable entera de tipo entero  
19 int *puntero; //Declaración de variable puntero de tipo entero  
20 puntero = &p; //Asignación de la dirección memoria de p  
21
```

```
22 printf("El valor de p es: %d. \nEl valor de *puntero es: %d. \n",p,*puntero);
```

```
23 printf("La dirección de memoria de *puntero es: %p. \n",puntero);
```

```
24 ++p=*puntero;
```

```
25 printf("El valor de p es: %d. \nEl valor de *puntero es: %d. \n",p,*puntero);*/
```

```
26 //falla porque del lado izquierdo debe haber un lvalue
```

```
27
```

```
28 //otro ejemplo
```

```
29 /*int i,j,z,y;
```

```
30 i=0; j=2;
```

```
31 (i<j?z:y)=4;
```

```
32 */
```

**Error: se requiere del lado izq de la asignación un l-valor pero lo que devuelve es un r-valor.**  
**++p es una expresión (p=1) (idem daría con p++)**

Para ejecutar sacar comentarios /\*

**error: se requiere lvalue como operando izquierdo de la asignación pero hay “expresión condicional”**

en la forma: **a ? b : c;**

**? operador condicional** (simil if/else.)

**a** es una *expresión booleana*, **b** y **c** pueden ser *expresiones o sentencias*.

Si el valor de **a** es verdadero se devuelve el valor de **b**, Si el valor de **a** es falso se devuelve el valor de **c**.

Ej. Si I es menor que J devuelve el **valor de Z** caso contrario **devuelve el valor de Y (el r-valor y no el l-valor)**

# SECUENCIA - SENTENCIA DE ASIGNACIÓN

## DISTINCIÓN EN ENTRE SENTENCIA DE ASIGNACIÓN Y EXPRESIÓN

### Resumen del código anterior:

#### Está permitido:

- $a=b=c=0;$                             C se evalúa de derecha a izquierda)
- if ( $i=30$ ) printf("Es verdadero")      asigna y evalúa

#### No está permitido:

- $++ p = *q;$
- $(i < j ? z : y) = 4;$
- La *mayoría de los lenguajes* de programación requieren que sobre el *lado izquierdo de la asignación* aparezca un *l-valor* y no un *r-valor*

C no permite una expresión  
del lado izquierdo que no denote un l-valor

## 2 SENTENCIA - SELECCIÓN - IF

- **IF** Estructura de control que permite expresar una elección entre un cierto número posible de sentencias alternativas (*ejecución condicional*)
- Entre los lenguajes la semántica es similar, en cuanto a la sintaxis existen muchas diferencias

- **Evolución:**

- **If lógico de Fortran**

**If** (condición lógica) sentencia

- Si la **condición es verdadera ejecuta la sentencia**
- **Esto permite tomar 1 camino posible**

## 2 SENTENCIA - SELECCIÓN - IF EN ALGOL

- **if then else de ALGOL**

**if** (condición lógica) **then** sentencia1  
**else** sentencia2

**Esto permite tomar 2 caminos posibles**

- El **problema if then else anidados.....**

## 2 SENTENCIA - SELECCIÓN - IF EN ALGOL

- **if then else anidados en Algol**

Con múltiples caminos posibles:

*if cond1 then if cond2 then if cond3 then .....*

- **Problema de Ambigüedad en Algol:**

El lenguaje no establecía cómo se asociaban los else con los if abiertos

*if  $x > 0$  then if  $x < 10$  then  $x := 0$  else  $x := 1000$*

¿Qué valor me devuelve X?

X inicial	Caso A (else al 2do if)	Caso B (else al 1er if)
10	Cambia 1000	Deja sin cambio 10
0	Deja sin cambio 0	Cambia 1000

## 2 SENTENCIA - SELECCIÓN - IF

- **Formas de eliminar la Ambigüedad:**  
**if then else caso de C, Pascal y Pl/1**

Regla 1: que cada **else** se empareje con la instrucción **if** solitaria más próxima buscando de derecha a izquierda.

**Cada else se empareja para cerrar  
al último if abierto**

Elimina la **ambigüedad**, pero las **instrucciones anidadas** pueden ser **difíciles de leer (*legibilidad, claridad*)**, especialmente si el programa está **escrito sin respetar sangría**.

## 2 SENTENCIA - SELECCIÓN - IF

### Problemas de claridad y legibilidad

**Regla 2:** Usar sentencias de cierre explícitas del bloque condicional if

por ejemplo

- **fi** en Algol 68
- **end if** en Ada
- **end Modula-2**
- No lo incluyen todos los lenguajes<sup>{}</sup>, pero hay formas de hacerlo por ejemplo con

Ejemplo: **Modula-2**

- *if x>0 then if x<10 then x:=0 else x:=1000 end end*
- *if x>0 then if x<10 then x:=0 end else x:=1000 end*

```
if i = 0
  then j := j;
  else i := i + 1;
      j := j - 1;
end
```

## 2 SENTENCIA - SELECCIÓN - IF

### Regla 3: sentencias compuestas (bloque de instrucciones)

- No todos los lenguajes lo permiten.
- El delimitador lo define cada uno.
- Algol 60 usa un **bloque de instrucciones** acotadas entre las palabras **begin** y **end**.

### Desventajas de if anidados:

**Ilegibilidad:** programas con muchos **if anidados** pueden ser **ilegibles, ambiguos, y difíciles de mantener.**

If expr1 then

Begin

If expr2 then  
a:=b+c

Else  
a:=b-c

End;

## 2 SENTENCIA - SELECCIÓN - IF EN C

### if then else en C en sentencia línea y en bloque de instrucciones

```
If (condición) Instrucción 1;  
else Instrucción A;
```

- Usa ()
- No usa then

```
if (condición) {  
    Instrucción 1;  
    Instrucción 2;  
    -  
    -  
    -  
    Instrucción n;  
}  
else {  
    Instrucción A;  
    Instrucción B;  
    -  
    -  
    -  
    Instrucción Z;  
}
```

- Usa {}
- No usa begin y end

#### Diferencias:

- C no lleva la palabra clave “then” como en Pascal
- Pascal no usa () en la condición

Se recomienda usar siempre llaves porque genera un código<sup>20</sup> más legible y más fácil de mantener, quedando bien delimitada la intención del programador.

## 2 SENTENCIA -

### SELECCIÓN CON EXPRESIÓN CONDICIONAL CORTA EN C

#### if corto en C

Se puede representar con expresión condicional:

**cond ? b : c;**      (i<j?z:y)

- **? operador condicional (operador ternario)**
- **cond** es una *expresión booleana*
- **b** y **c** pueden ser *expresiones o sentencias*.
- Si el valor de **cond** es verdadero se **devuelve el valor de b**
- Si el valor de **cond** es falso se **devuelve el valor de c**

...

#### Pasar de if largo a if corto

```
country = ''  
if(lang == 'es') {  
    country = 'ES'  
} else {  
    country = 'EN'  
}
```

A algo como esto,

```
...  
country = (lang == 'es' ? 'ES' : 'EN')
```

## 2 SENTENCIA - SELECCIÓN - IF EN PYTHON

### If condición then ..... else de Python

- incorpora **elif** si hay más de 2 opciones y **sangría**
- elimina ambigüedad y da legibilidad

```
if sexo == 'M':  
    print 'La persona es Mujer'  
else:  
    print 'La persona es de otro género'
```

```
if hora >=4 and hora <= 12:  
    print 'Buenos días!!'  
elif hora >12 and hora < 20:  
    print 'Buenas tardes!!'  
else:  
    print 'Buenas noches!!'
```

- **:** – obligatorio al final del **if**, **else** y **elif**
- Indentación – obligatoria tanto en **if**, **elif** y **else** (4 espacios en blanco sin esto finaliza el bloque de código)
- **( )** – opcional su uso en la **condición**

## 2 SENTENCIA -

### SELECCIÓN CON EXPRESIÓN CONDICIONAL CORTA EN PYTHON

[on\_true] **if** [expresión] else [on\_false] de **Python**

- **Expresión condicional con** "operador ternario" o "operador condicional"
- **Construcción equivalente al “?” del lenguaje C**

$$\begin{array}{c} \text{C ? A : B} \quad \text{(en C)} \\ \hline \text{A if C else B} \quad \text{(en Python)} \end{array}$$

Devuelve A si se cumple la condición C, sinó devuelve B

```
>>> altura = 1.79
>>> estatura = "Alto" if altura > 1.65 else "Bajo"
>>> estatura
'Alto'
>>>
```

Python, C para evaluar las **condiciones** utiliza la<sup>24</sup>  
“Evaluación con circuito corto” y de izquierda a derecha

# EVALUACIÓN DE LAS EXPRESIONES EN LOS LENGUAJES

La terminología "circuito corto" y "circuito largo" se refiere a cómo se evalúan las expresiones lógicas en ciertos contextos. Son técnicas utilizadas en la **evaluación de expresiones booleanas** que involucran **operadores lógicos tipo *AND* y *OR*** (*no aplica a XOR*).

- **Circuito corto:** Significa que **los operandos de una expresión lógica se evalúan de izquierda a derecha, pero solo hasta encontrar el primero que determina el resultado final**. En ese punto, la evaluación se detiene y no se evalúan los operandos restantes.
- **Circuito largo:** Significa que **todos los operandos se evalúan, sin importar si el primero ya determina el resultado final**.

# CIRCUITO CORTO

TAMBIÉN CONOCIDA COMO "EVALUACIÓN PEREZOSA"  
O "EVALUACIÓN DE CORTOCIRCUITO"

Se trata de una forma de evaluar expresiones lógicas. Se implementa en muchos lenguajes de programación.

**La conjunción ("y" / "and") da como resultado verdadero únicamente cuando ambos términos son verdaderos. Si el primer término es falso, no es necesario evaluar el segundo: el resultado será falso.**

**La disyunción ("o" / "or") da como resultado falso únicamente cuando ambos términos son falsos. Si el primer término es verdadero, no es necesario evaluar el segundo: el resultado será verdadero.**

Ejemplo en  
Python

```
def a(): print("a"); return True
def b(): print("b"); return True
def c(): print("c"); return False
def d(): print("d"); return True
```

```
if a() and b() and c() and d():
    print("Todo pasó")
else:
    print("Fallo")
```



# CIRCUITO CORTO

## EL CIRCUITO CORTO PERMITE EVITAR ERRORES Y OPTIMIZAR EL RENDIMIENTO

### Ventajas - Evitamos:

- *Errores por división por cero*
- *Errores por acceso a punteros nulos*
- *Errores por índice fuera de rango* (IndexError)
- *Errores por acceso objetos nulos* (null reference)
- *Errores lógicos* o efectos no deseados al *invocar funciones* que no deberían ejecutarse
- *Consumo de tiempo* innecesario en *operaciones y chequeos costosos*

```
#include <stdio.h>
```

```
int x = 0;
```

```
int division_segura() {  
    printf("Intentando dividir...\n");  
    return (x != 0) && (10 / x); // No se evalúa (10 / x) si x == 0  
}
```

```
int main() {  
    if (division_segura()) {  
        printf("División exitosa\n");  
    } else {  
        printf("División no realizada  
(evitada)\n");  
    }  
    return 0;  
}
```

Usamos x para garantizar que la división no será nula

# LENGUAJES QUE IMPLEMENTAN CIRCUITO CORTO

Lenguaje	Operadores de corto circuito	Evaluación izquierda → derecha	Lenguaje	Operadores de corto circuito	Evaluación izquierda → derecha
C	&&,	☑	PHP	&&,	☑
C++	&&,	☑	Haskell	&&,	☑
Java	&&,	☑	Ada	and then, or else	☑
JavaScript	&&,	☑	Pascal	and, or	✗
Python	and, or	☑	Modula-2	AND, OR	✗
Ruby	and, or	☑	Fortran	.AND., .OR.	✗
Go	&&,	☑	Lisp (Common Lisp)	and, or	☑
Swift	&&,	☑	Scala	&&,	☑
Kotlin	&&,	☑	R	&&,    (escalares)	☑
Rust	&&,	☑	Bash / Shell	&&,    (comandos)	☑



## 2 SENTENCIA - SELECCIÓN MÚLTIPLE

- Los lenguajes incorporan *distintos tipos de sentencias de selección múltiple* para poder elegir entre dos o más **opciones/caminos posibles**, y salir de IF anidados.

**Para reemplazar a estructuras del tipo:**

```
if (A) then sentencia1  
        else if (B) then sentencia2  
                else if .....  
                else sentencia n;
```

Los lenguajes tienen su propia sintaxis y su propia forma de implementarlo.

Veamos ejemplos.....

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - PASCAL

- Usa **palabra reservada case** seguida de **variable-expresión de tipo ordinal** y la **palabra reservada of**.
- **Variable-expresión** a evaluar es llamada “**selector**”
- **Lista de sentencias** para los diferentes valores que puede adoptar la variable (**los “casos”**), que además llevan **etiquetas**.
- **No importa el orden**
- **bloque else** para el caso que la variable adopte un valor que **no coincide con ninguna de las sentencias de la lista.** (**opcional**)
- Para **finalizar** se coloca un **end;** (no se corresponde con ningún “begin” que exista).

### Sintaxis:

```
case variable_ordinal of
    valor1: sentencia 1;
    valor2: sentencia2;
    valor3: sentencia3;
else
    sentencia4;
end;
```

**Ordinal:** puede obtenerse un predecesor y un sucesor (a excepción del primer y el último (expresa la idea de orden o sucesión)

**Inseguro:** al no establecer que sucede cuando un valor no cae dentro de ninguna de las alternativas.

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - PASCAL

else es **opcional**. ---- Inseguro

¿Qué sucede si se ingresa un número y no hay un else?

- La "no acción" puede **ocultar errores lógicos graves, comportamientos inesperados**, variables sin definir, funciones sin ejecutar, etc.
- **Algunos compiladores no advierten si no se cubren todos los casos posibles**
- El programador debe ocuparse de considerar todas las opciones

Ejemplo:

```
var opcion : char;  
begin  
    readln(opcion);  
    case opcion of  
        '1' : nuevaEntrada;  
        '2' : cambiarDatos;  
        '3' : borrarEntrada;  
        else  
            writeln('Opcion no valida!!')  
    end;  
end
```

**Doble end**  
por el **begin**  
que ya existe

else cambió  
entre versiones  
de Pascal,  
otherwise

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - ADA

Constructor:

**Case expresión is**  
**con when y end case**

- Tiene **reglas más estrictas**.
- Las **expresiones sólo** pueden ser de tipo **entero o enumerativo**
- El **case debe contemplar todos los valores posibles** que puede tomar la expresión. *Sino error del compilador*
- El **when =>** para indicar la acción/ sentencia a ejecutar si se cumple la condición.
- **end case; debe** ser siempre el **cierre final**

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - ADA

- **When Others** se puede utilizar para representar a aquellos valores que no se especificaron explícitamente. (opcional)
- **When Others debe** ser la última opción antes del **end case;**
- Una vez que se ejecuta una rama del **case**, el **case** finaliza y no se ejecuta ninguna otra rama.

**No pasa la compilación si:**

NO se coloca la rama para un posible valor y  
NO aparece la opción **Others** en esos casos

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - ADA

### Ejemplo 1

```
case Operador is
    when '+' => result:= a + b;
    when '-' => result:= a - b;
    when others => result:= a * b;
end case;
```

Importante para el programador:

**others** se debe colocar porque las etiquetas de las ramas NO abarcan todos los posibles valores de Operador

### Ejemplo 2

```
type Dia_Semana is (Lunes, Martes, Miércoles);
Dia : Dia_Semana;
```

```
case Dia is
    when Lunes => ...
    when Martes => ...
end case;
```

**ERROR**  
compilación x  
miércoles

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - ADA

### Ejemplo 23

**case** Hoy **is**

**when** MIE..VIE => Entrenar\_duro;

-- *Se puede especificar Rango con ..*

**when** MAR | SAB => Entrenar\_poco;

-- *Se puede especificar varias elecciones |*

**when** DOM => Competir;

-- *Única elección.*

**when others** => Descansar;

-- *Debe ser única y la última  
alternativa. (LUN)*

**end case;**

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE – C, C++

- Constructor **Switch (expresión)**
- Cada rama **Case** es "etiquetada" por uno o más **valores constantes (enteros o char)**
- Si coincide con una etiqueta se ejecutan las sentencias asociadas, y continúa con las sentencias de las otras entradas. (*chequea todas salvo exista un break*)
- La sentencia **break** (opcional) provoca la salida de cada rama
- La sentencia **default** (opcional) para los casos en que el valor no coincide con ninguna de las opciones establecidas
- El orden en que aparecen las ramas **no tiene importancia**<sup>38</sup>

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE – C, C++ "FALLING THROUGH"

El **control de flujo** pasa de un **case a otro** consecutivo cuando se omite el **break** después de un case.

### Funcionamiento del "Falling Through":

Cuando se evalúa una expresión en un switch-case en C, el **compilador busca un case cuyo valor coincide con el valor de la expresión**. Una vez que encuentra un case con coincidencia, **ejecuta el código** asociado a ese case y luego **continúa ejecutando** el código de los **case siguientes** que no tienen **break** hasta encontrar un **break**, **Default** o llegar al final del **switch**.

```
opcion = 1:  
switch (opcion) {  
    case 1:  
        printf("Opción 1\n");  
        // Falta el break, por lo tanto, se ejecutará también el siguiente case  
    case 2:  
        printf("Opción 2\n");  
        break;  
    case 3:  
        printf("Opción 3\n");  
        break;  
    default:  
        printf("Opción por defecto\n");  
        break;  
}
```

Imprime  
"Opción 1" y  
"Opción 2"

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE – C, C++

### "FALLING THROUGH"

#### Desventajas del "Falling Through":

- Puede producir errores. La **falta de break** puede llevar a **errores lógicos difíciles de detectar**.
- Puede generar **comportamientos indefinidos**. La **falta de default** puede llegar a causarlo.
- Puede **reducir la claridad y legibilidad** del código al no estar **explícito el final de cada bloque**.
- Requiere disciplina para **documentar**.
- Puede generar **bugs de mantenimiento** si intervienen **distintos programadores**.
- **No todos los compiladores alertan la falta de un break.**

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE – C, C++

**Switch (Operador) {**

**case ' + ' :**

    result:= a + b; **break;**

**case ' - ' :**

    result:= a - b; **break;**

**default : //Opcional**

    result:= a \* b;

}

Conviene usar  
**break** para saltar  
las siguientes  
ramas, **sinó pasa**  
**por todas**

**switch( i )**

{

**case -1:**

    n++;

**break;**

**case 0 :**

    z++;

**break;**

**case 1 :**

    p++;

**break;**

}

Para evitar problemas y asegurar comportamiento, es buena práctica siempre tener en cuenta todas las posibles entradas que puede recibir la expresión del switch y manejarlas adecuadamente con case específicos y un default si es necesario.

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - RUBY

- Constructor **Case expresión**, seguido de **when** y **end**
- La **expresión** es **cualquier valor (cadena, número o expresión que proporcione algunos resultados como "a", 1 == 1, etc.)**.
- Se buscará la expresión en los **bloques when** hasta que **coincida con la condición**, y **ejecutará ese bloque de código**.
- Si **no coincide con ninguna expresión** irá al bloque **else** (equivalente a Default).
- **else** es **opcional**, esto puede traer **efectos colaterales**

**Se recomienda programación defensiva:**

**La cláusula debe tomar la acción apropiada o**  
**contener un comentario adecuado sobre por qué no se**  
**toma ninguna acción.**

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE – RUBY

### Noncompliant Code Example

```
case param
  when 1
    do_something()
  when 2
    do_something_else()
end
```

Si no entra en ninguna opción, y sigue la ejecución y la variable no se le asignara ningún valor.

Podría llevar a error

### Compliant Solution

```
case param
  when 1
    do_something()
  when 2
    do_something_else()
  else
    handle_error('error_message')
end
```

Consejo de programación defensiva:

La cláusula debe tomar la acción apropiada o contener un comentario adecuado sobre por qué no se toma ninguna acción.

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE – RUBY

### El Señor de los Anillos

```
raza = 'enano'

# Ahora utilizaremos el case
puts 'Utilizando case asignado a una variable :'
personaje = case raza
  when 'elfo' then 'Legolas'
  when 'enano' then 'Gimli'
  when 'mago' then 'Gandalf'
  when 'ents' then 'Barbol'
  when 'humano' then 'Aragorn'
  when 'orco' then 'Ufthak'
end
puts personaje
```

**ERROR:** Si no entra en ninguna opción, sigue la ejecución y la variable no tendrá ningún valor!

## 2 SENTENCIA - SELECCIÓN MÚLTIPLE - PL/I

- Sentencia **SELECT** de PL/1 incorpora el uso de **WHEN/OTHERWISE/END**
- Tiene 2 tipos de formatos: **Identificador o Condición**

-----Formato - 1-----

```
SELECT (identifier) ;  
      WHEN (value1) statement;  
      WHEN (value2) statement;  
      OTHERWISE statement ;  
END;
```

-----Formato - 2 -----

```
SELECT ;  
      WHEN (cond1) statement;  
      WHEN (cond2) statement;  
      OTHERWISE statement ;  
END;
```

### 3 ITERACIÓN

- La **iteración** permite que una **serie de acciones se ejecuten repetidamente (loop)**.
- La mayoría de los lenguajes de programación proporcionan **diferentes tipos de construcciones de bucle** para definir la *iteración de acciones (llamado el cuerpo del bucle)*.
- Comúnmente agrupados como :
  - **bucle for:** bucles en los que *se conoce el número de repeticiones al inicio del bucle.* (se repiten un cierto número de veces)
  - **bucle while:** bucles en los que el cuerpo se *ejecuta repetidamente siempre que se cumpla una condición.* (hay 2 tipos)

### 3 ITERACIÓN - SENTENCIA DEL TIPO FOR LOOP

#### □ Sentencia **Do** de Fortran

**Do** **label** **var-de-control**= **valorIni**, **valorFin**

.....

**label** **continue**

- La **var-de-control** **solo** puede tomar **valores enteros** y se **incrementa 1 en 1 en la secuencia** (*si no se especifica otra cosa según lo permita el lenguaje*)
- **ValorIni** es el valor inicial
- **ValorFin** es el valor final
- La **declaración** **continue** junto con la **etiqueta** **label** se usa como la **última declaración de un DO**<sup>47</sup> (*depende versión de FORTRAN*)

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

#### □ Sentencia Do de Fortran

- Fortran original evaluaba si la variable de control había llegado al límite al final del bucle al final (*"siempre una vez lo ejecutaba" y traía problemas*)
- Desde FORTRAN 77 se evalúa antes
- La **variable de control "nunca" deberá ser modificada por otras sentencias dentro del ciclo**, ya que puede generar errores de lógica.

Ejemplos:

```
DO 1 I = 1,10  
      SUM = SUM A(I)  
1    CONTINUE
```

```
DO 1 I = 10,1  
      SUM = SUM A(I)  
1    CONTINUE
```

El **2do no ejecuta**  
porque  
**valorIni es mayor**  
**que valorFin**

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

#### Sentencia For de Pascal, ADA, C , C++

**for loop\_ctrl\_var := lower\_bound to upper\_bound do statement**

- La **variable de control** puede tomar **cualquier valor ordinal (enumerativos)** que indiquen secuencia, *no sólo enteros*
- Pascal estándar "no permite" que se **modifiquen** los **valores** del **límite inferior**, **límite superior**, ni del **valor de la variable de control**.
- El **valor de la variable fuera del bloque** se **asume no definida**

# EJEMPLO EN PASCAL - MODIFICACIÓN DE LA VARIABLE DE CONTROL

```
1
2 Program HelloWorld(output);
3 Uses sysutils;
4 var i: Integer;
5 Procedure VerI();
6 Begin
7   writeln(Concat('valor de i es ',IntToStr(i)));
8   // es inseguro si tocamos el iterador en otra unidad.
9   //i:=i+20; ←
10 end;
11 BEGIN
12   writeln('Hello, world!');
13   for i:=1 to 20 do begin
14     writeln(Concat('valor de i es ',IntToStr(i)));
15     IF (i=5) THEN begin
16       VerI();
17     end
18     ELSE begin
19       //no se permite alterar el iterador en la misma unidad
20       //i:=i+1; ←
21     end;
22   end
23 END.
```

## Salida del programa

Free Pascal Compiler version 3.2.0  
**valor de i es 1.....20**

si **descomento i:=i+1** da este **error**  
Compiling main.p, main.p(22,8)  
**Error: Illegal assignment to for-loop variable "i" main.p(25,4)**  
Fatal: There were 1 errors compiling module, stopping  
**Fatal: Compilation aborted**  
Error: /usr/bin/ppcx64 returned an error exitcode (**no permite modificar la variable en el loop**)

si **descomento i=1+20** da  
Hello, world!  
**valor de i es 1 2 3 4 5 5**  
**Puedo modificar fuera la variable hay efecto colateral**

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

#### Sentencia For de ADA

- Encierra todo proceso iterativo entre las cláusulas **loop** y **end loop**.
- Permite el uso de la sentencia **Exit** para **salir del loop**.
- La **variable de control (iterador)** es de **Tipo enumerativa**
- La variable de control NO necesita declararse (*se declara implícitamente al entrar al bucle y desaparece al salir*).
- El **in** indica **incremento**, permite **decrementar** con **in reverse**

**for i in 1..N loop**

$V(i) := 0;$

**end loop**

**for i in reverse 1..N loop**

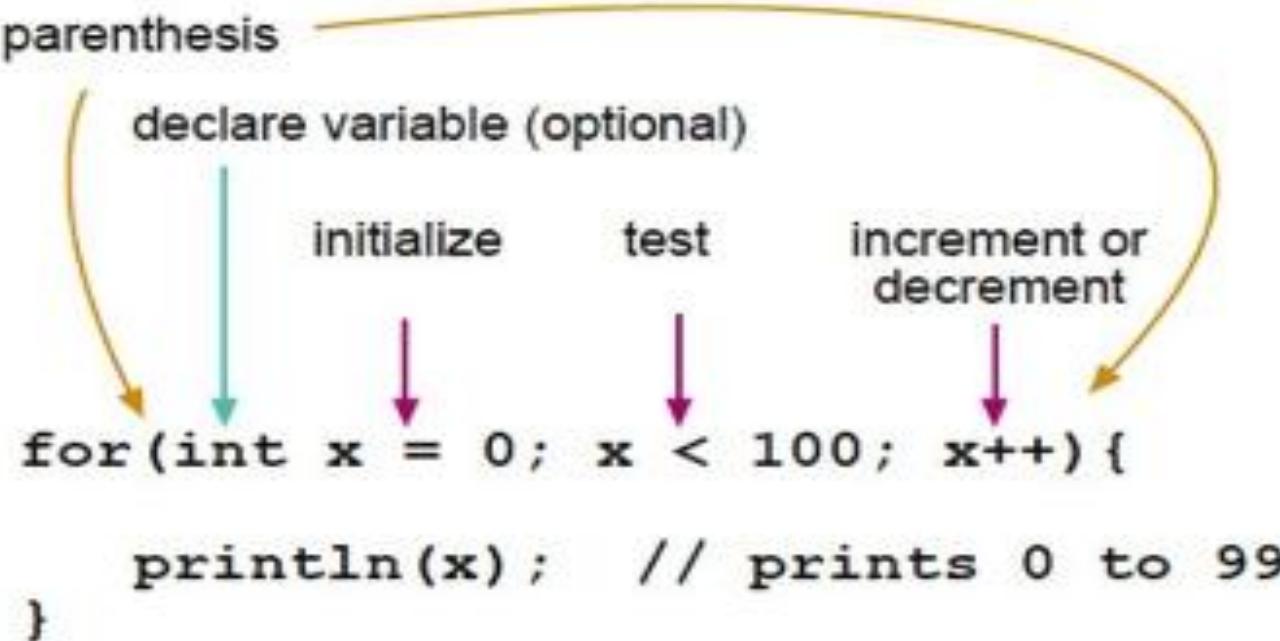
$V(i) := 0;$

**end loop**

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

#### Sentencia For de C, C++

- Se componen de **3 partes**: 1 **inicialización** y 2 **expresiones**
- **Inicialización**: da el **estado inicial** para la ejecución del bucle.
- **1ra expresión**: especifica el **test** que es **realizado antes de cada iteración**. Sale del ciclo si la expresión no se cumple (sale del rango, i.e false)
- **2da expresión**: especifica el **incremento** que se **realiza después de cada iteración**.



### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

#### Sentencia **For** de C++

- En **C++** se puede realizar la **declaración** de una **variable** dentro del **for (....)**
- El alcance de la variable se extiende hasta el final del bloque que encierra la instrucción **for {....}**
- Si se **omiten una o ambas expresiones** en un bucle for se puede **crear un bucle sin fin**, del que **solo se puede salir con una instrucción break, goto o return.**

```
for ( ; ; ) { . . . }
```

Loop infinito

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

#### Sentencia **For** de **Python**

##### Phyton tiene 2 tipos de estructuras For:

- para iterar sobre una secuencia de estructuras de datos de tipo:
  - *lista, tupla, conjunto, diccionario, etc.*
- para iterar sobre un rango de valores basado en una secuencia numérica:
  - usando **función Range()**

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

Sentencia **For** de **Python** para iterar en una secuencia

```
for <elem> in <iterable>:  
    <codigo>
```

- El bucle **for** para recorrer todos los elementos de un objeto iterable (*lista, tupla, conjunto, diccionario, ...*) y ejecutar un bloque de código.
- **elem** variable que toma el valor del elemento dentro/**in** del iterador **iterable** en cada paso del bucle.
- En cada paso de la iteración se tiene en cuenta a un único elemento del objeto iterable, sobre el cuál se pueden aplicar una serie de operaciones.
- Finaliza su ejecución cuando se recorren todos los elementos.

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

Sentencia **For** de **Python** para iterar en una secuencia

```
lista = [ "el", "for", "recorre", "toda", "la", "lista"]
for variable in lista:
    print variable
```

Imprimirá:

el  
for  
recorre  
toda  
la  
lista

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

## Sentencia **For** de **Python** para iterar sobre un rango de valores

- Uso de la *funcion/clase* **range(max)**
- que **devuelve** valores van desde **0** hasta **max - 1**.
- De esta forma el **for** actúa como los demás lenguajes

```
1. for i in range(11):  
2.     print(i)  
3.  
4. 0  
5. 1  
6. 2  
7. 3  
8. ...  
9. 10
```

### 3 ITERACIÓN - SENTENCIA TIPO FOR LOOP

## Sentencia **For** de **Python** para iterar sobre un rango de valores

El tipo de datos `range` se puede invocar con uno, dos e incluso tres parámetros:

- `range(max)`: Un iterable de números enteros consecutivos que empieza en `0` y acaba en `max - 1`
- `range(min, max)`: Un iterable de números enteros consecutivos que empieza en `min` y acaba en `max - 1`
- `range(min, max, step)`: Un iterable de números enteros consecutivos que empieza en `min` acaba en `max - 1` y los valores se van incrementando de `step` en `step`. Este último caso simula el bucle `for` con variable de control.

Por ejemplo, para mostrar por pantalla los números pares del 0 al 10 podríamos usar la función `range` del siguiente modo:

```
1. for num in range(0, 11, 2):  
2.     print(num)
```

- |    |                             |
|----|-----------------------------|
| 1. | for num in range(0, 11, 2): |
| 2. | print(num)                  |
| 3. |                             |
| 4. | 0                           |
| 5. | 2                           |
| 6. | 4                           |
| 7. | 6                           |
| 8. | 8                           |
| 9. | 10                          |
- 1 argumentos: `range(5)` **RANGO MAX** devuelve [0,1,2,3,4]
  - 2 argumentos: `range(2,5)` **RANGO MIN-MAX** devuelve [2,3,4]
  - 3 argumentos: `range(2,5,2)` **RANGO Y STEP** devuelve [2,4]

### 3 ITERACIÓN - SENTENCIA TIPO WHILE LOOP (CHEQUEO A INICIO)

#### while

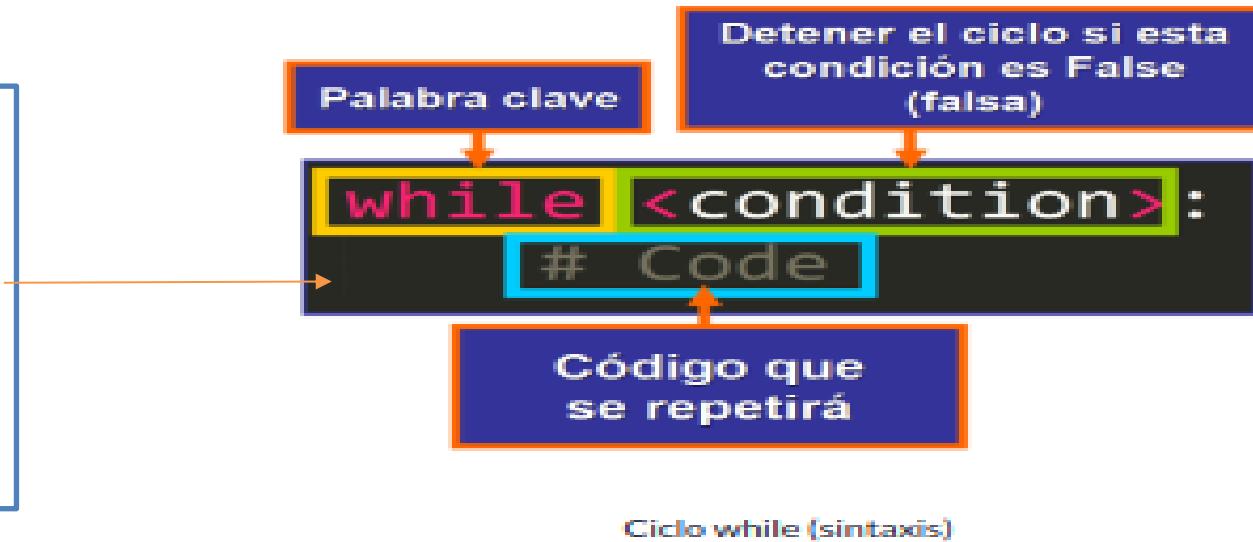
- Estructura que permite **repetir un proceso** mientras se cumpla una **condición**.
- La condición **se evalúa antes** de que se entre al proceso

PASCAL	<code>while condición do sentencia begin .... end</code>
C, C++	<code>while (condición) sentencia; {....}</code>
ADA	<code>while condición sentencia .... end loop;</code>
PYTHON	<code>while condición :     Sentencia 1     Sentencia 2     ....     sentencia n</code>

# 3 ITERACIÓN - SENTENCIA TIPO WHILE LOOP (CHEQUEO AL INICIO)

## While de PYTHON

La guía de estilo de Python (PEP 8) recomienda usar 4 espacios por nivel de indentación (sangría).



Los principales elementos son:

- la **palabra clave** **while** (seguida de un espacio)
- una **condición** que determina si el ciclo **continuará su ejecución o no** en base a su valor (**true o false**)
- **:** al **final de la primera línea**
- la secuencia de instrucciones o sentencias que se repetirán. A este bloque de código se lo denomina el "**cuerpo**" del **ciclo** y debe estar **indentado**. **Si una línea de código no está indentada no se lo considerará parte del ciclo**

### 3 ITERACIÓN - SENTENCIA TIPO WHILE LOOP (CHEQUEO A FINAL)

#### Until-Repeat y Do-While

- Estructuras que permite repetir un proceso "hasta" que se cumpla una condición. En definitiva, permite ejecutar un bloque de instrucciones *mientras no se cumpla una condición dada (o sea falso)*.
- La condición/expresión se evalúa al final del proceso, por lo que *por lo menos 1 vez el proceso se realiza*

PASCAL	<b>repeat</b> Sentencia ..... <b>until condición;</b>
C, C++	<b>do</b> sentencia; ..... <b>while (condición);</b>

### 3 ITERACIÓN - SENTENCIA LOOP

**ADA usa una estructura iterativa**

Para lograr un comportamiento similar a un bucle "while until" en Ada se usa:

```
[ identificador_bucle : ] loop  
    secuencia_de_sentencias  
end loop [ identificador_bucle ] ;
```

Se puede combinar con estructuras FOR y WHILE

```
Vida: loop -- El bucle dura indefinidamente.  
    Trabajar;  
    Comer;  
    Dormir;  
end loop Vida;
```

### 3 ITERACIÓN - SENTENCIA LOOP ADA

- De este bucle se **sale** normalmente, mediante una **sentencia "exit when condition"**

```
loop
  ....
  exit when condición;
  ...
end loop;
```

```
loop
  Alimentar_Caldera;
  Monitorizar_Sensor;
  exit when Temperatura_Ideal;
end loop;
```

- O con una **alternativa** que contenga una cláusula "**exit<sup>63</sup>**"

# CONCLUSIÓN

Los lenguajes tienen sintaxis muy distintas para las estructuras de control, deben ser conocidas por el programador

- **Entender los fundamentos:** es crucial comprender los conceptos fundamentales de las estructuras de control, como los bucles, las instrucciones condicionales y los saltos de control. Esto incluye comprender cómo funcionan, cuándo usar cada tipo de estructura y cómo diseñar algoritmos efectivos utilizando estas estructuras.
- **Practicar y experimentar:** familiarizarse con la sintaxis y las estructuras de control, practicar escribiendo código y experimentar con diferentes construcciones. Así se aprende a utilizar las estructuras de control de manera efectiva.