Apuntes de Programación.

Departamento de Informática.
Colegio Salesiano Santo Domingo Savio

10 de octubre de 2011

Palabras Clave: libro de programación, c, c++, programación, informática, computadores, sintáxis, pseudocódigo
Clasificación ACM: D.3.3. Language Constructs and features.

Copyright ©2011 Colegio Salesiano Santo Domingo Savio. Se concede permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre de GNU, Versión 1.2 o cualquier otra versión posterior publicada por la Free Software Foundation. Una traducción de la licencia está incluida en la sección titulada "Licencia de Documentación Libre de GNU".
Copyright ©2011 Colegio Salesiano Santo Domingo Savio. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Ver- sion 1.2 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled "GNU Free Documentation License".

Índice general

I pseudocodigo 5

1. Datos 6
 1.1. Tipos de datos . . . 6
 1.2. Variables y Constantes. . . . 6
  1.2.1. Variables . . . 6
  1.2.2. Constantes . . . 8

2. Operadores 9
 2.1. Operadores aritméticos . . . 9
 2.2. Operadores Relacionales . . . 9
 2.3. Operadores Lógicos . . . 10
 2.4. Operadores a Nivel de Bits . . . 10
 2.5. Operadores taquigráficos . . . 11

3. Estructura de un Programa 12

4. Instrucciones y Estructuras de Control 13
 4.1. Primitivas de Entrada, Salida y Asignación . . . 13
  4.1.1. Entrada . . . 13
  4.1.2. Salida . . . 13
 4.2. Asignación . . . 14
 4.3. Bloques . . . 14
 4.4. Alternativas . . . 15
  4.4.1. Condicional simple . . . 15
  4.4.2. Condicional compuesta . . . 16
  4.4.3. Condicional múltiple . . . 17
 4.5. Iterativas . . . 19
  4.5.1. Mientras . . . 19
  4.5.2. Repetir . . . 20
  4.5.3. Desde . . . 21

5. Programación modular 24
 5.1. Funciones . . . 24
 5.2. Procedimientos . . . 26

II C **28**

6. Introducción **29**
6.1. Los Cinco Mandamientos del Programador 29
6.2. Los componentes del lenguaje 30
6.3. Los Componentes Sintácticos 31
6.3.1. identificadores 31
6.3.2. Cadenas de Caracteres 32
6.3.3. Comentarios 32

7. Datos **34**
7.1. Tipos de datos 34
7.2. Variables y Constantes. 34
7.2.1. Variables 34
7.2.2. Constantes 39

8. Operadores **45**
8.1. Operadores aritméticos 45
8.2. Operadores Relacionales 45
8.3. Operadores Lógicos 46
8.4. Operadores a Nivel de Bits 46
8.5. Operadores taquigráficos 47

9. Estructura de un Programa **48**

10. Instrucciones y Estructuras de Control **49**
10.1. Primitivas de Entrada, Salida y Asignación 49
10.2. Asignación 49
10.2.1. Salida 50
10.2.2. Entrada 51
10.3. Bloques 52
10.4. Alternativas 52
10.4.1. Condicional simple 53
10.4.2. Condicional compuesta 54
10.4.3. Condicional múltiple 55
10.5. Iterativas 57
10.5.1. while 57
10.5.2. do while 58
10.5.3. for 60

11. Programación modular **62**
11.1. Funciones 62
11.1.1. Declaración 63
11.1.2. Definición 63
11.1.3. Llamada 64

11.1.4. La función main 67
11.1.5. Paso por Valor y Paso por Referencia 68
11.1.6. Recursividad 70
11.1.7. Ámbito y Visibilidad 71
11.1.8. Punteros a funciones 72

Parte I

pseudocodigo

Capítulo 1

Datos

1.1. Tipos de datos

Los tipos de datos permitidos para pseudocódigo son:

ENTERO Representa valores enteros positivos y negativos.

REAL Representa valores reales con 35 dígitos de precisión.

CARACTER Representa valores alfanuméricos.

LOGICO Representa valores lógicos (VERDADERO o FALSO).

1.2. Variables y Constantes.

Un dato es toda aquella información relevante que puede ser tratada con posterioridad en un programa. Según el modo de almacenamiento, existen 2 tipos de datos: variables y constantes.

1.2.1. Variables

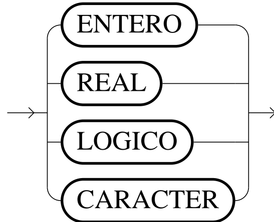
Declaracion_de_variable

→ [TIPO_DE_DATOS] → [identificador] →

```
<TIPO DE DATOS>   <identificador>;
```

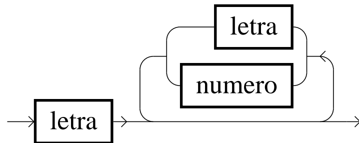
Donde:

```
TIPO DE DATOS = "ENTERO" | "REAL" | "CARACTER" | "LOGICO";
identificador = nombre;
```



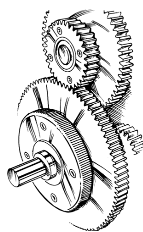
El identificador es un nombre cualquiera con el que se va a bautizar la variable. En general se recomienda:

```
identificador = minuscula { minuscula | mayuscula | numero };
minuscula     = "a" | "b" | "c" | "d" |
                "e" | "f" | "g" | "h" |
                "i" | "j" | "k" | "l" |
                "m" | "n" | "o" | "p" |
                "q" | "r" | "s" | "t" |
                "u" | "v" | "w" | "x" |
                "y" | "z" | "_" ;
```



Ejemplo:

```
ENTERO   numero;  # Definicion de la variable
                  # numero de tipo entero.
CARACTER letra;   # Definicion de la variable
                  # letra  de tipo caracter.
```



Las variables numéricas tienen uso como almacen de datos –lo importante es que guardan un dato–, como contadores –se incrementan generalmente de 1 en 1, pero lo puede hacer de n en n, $n \in \mathbb{R}$– o como acumulador – su valor se incrementa/decrementa en una cantidad–

1.2.2. Constantes

Declaracion_de_constantes

```
<TIPO DE DATOS>   <identificador> = <valor constante>;
```

Donde:

```
valor constante = constante numerica |
                  constante caracter;
```

Ejemplo:

```
REAL PI = 3.1416;     # Definicion de la constante PI
                      # cuyo valor durante todo el
                      # programa sera 3.141.6.
CARACTER LETRA = 'P' # Definicion de la constante LETRA
                      # cuyo valor durante todo el
                      # programa ser  'P'.
```

Capítulo 2

Operadores

Los operadores determinan las diferentes operaciones que se pueden realizar son los operandos. Los operadores se clasifican en:

2.1. Operadores aritméticos

Realizan una operación aritmética, los operandos siempre son de tipo numérico (ENTERO o REAL) y devuelven un resultado numérico (ENTERO o REAL).

Operador	Nombre	Lectura	Modo de Uso
+	Suma	más	$operando1 + operando2 \Rightarrow suma$
-	Resta	menos	$operando1 - operando2 \Rightarrow resta$
*	Multiplicación	por	$operando1 * operando2 \Rightarrow multiplicacion$
/	División	entre	$operando1/operando2 \Rightarrow division$
%	Resto	resto	$operando1\,\%operando2 \Rightarrow resto$

Ejemplo:

```
23    +  2    =>  25
 3    −  12   =>  −9
 7,5  ∗  2    =>  15
12,4  /  2    =>  6,2
 5    %  2    =>  1
```

2.2. Operadores Relacionales

Comprueban la relación existente entre dos operandos. Los operandos pueden ser de cualquier tipo siempre que ambos sean del mismo tipo. Devuelven un resultado lógico (VERDADERO ó FALSO).

Operador	Nombre	Lectura	Modo de Uso
==	igualdad	es igual a	$operando1 == operando2$
!=	desigualdad	es distinto de	$operando1 != operando2$
<	menor	es menor que	$operando1 < operando2$
<=	menor o igual	es menor o igual que	$operando1 <= operando2$
>	mayor	es mayor que	$operando1 > operando2$
<=	mayor o igual	es mayor o igual que	$operando1 >= operando2$

Ejemplo:

23 == 4 => **FALSO**
7 != 8 => **VERDADERO**
34 < 12 => **FALSO**
12 <= 12 => **VERDADERO**
4 > −2 => **VERDADERO**
−4 >= 1 => **FALSO**

2.3. Operadores Lógicos

Realizan una operación lógica. Los operandos deben ser de tipo lógico (VERDADERO ó FALSO) y devuelven un resultado lógico.

Operador	Nombre	Lectura	Modo de Uso
NOT	Negación	no	NOT operando
AND	Y Lógico	y	operando1 AND operando2
OR	O Lógico	o	operando1 OR operando2

Ejemplo:

NOT **VERDADERO** => **FALSO**
NOT **FALSO** => **VERDADERO**
VERDADERO AND **VERDADERO** => **VERDADERO**
VERDADERO AND **FALSO** => **FALSO**
FALSO AND **VERDADERO** => **FALSO**
FALSO AND **FALSO** => **FALSO**
VERDADERO OR **VERDADERO** => **VERDADERO**
VERDADERO OR **FALSO** => **VERDADERO**
FALSO OR **VERDADERO** => **VERDADERO**
FALSO OR **FALSO** => **FALSO**

2.4. Operadores a Nivel de Bits

Los operadores a nivel de bits toman cada uno de los bits de una cifra, cuando está expresada en binario, como si fuera un valor lógico.

Operador	Nombre	Lectura	Modo de Uso
&	Y	y	$operando1\&operando2$
$\vert$	O	o	$operando1\vert operando2$
$\sim$	no	no	$operando1 \sim operando2$
^	o exclusivo	xor	$operando1^\wedge operando2$
>>	desplazamiento	desplazado operando2 posiciones a la der.	$operando1 >> operando2$
<<	desplazamiento	desplazado operando2 posiciones a la izq.	$operando1 << operando2$

Ejemplo:

```
 6  &   5 => 4
    ~   5 => 2
 6  |   5 => 7
 6  ^   5 => 3
23 >>   2 => 5
 4 <<   1 => 8
```

2.5. Operadores taquigráficos

Los operadores taquigráficos no son esencialmente distintos de los operadores ya vistos, sino una manera abreviada de usarlos como acumulador.

Operador	Nombre	Modo de Uso	equivalencia
+=	Suma	$op1+ = op2$	$op1 = op1 + op2$
-=	Resta	$op1- = op2$	$op1 = op1 - op2$
*=	Multiplicación	$op1* = op2$	$op1 = op1 * op2$
/=	División	$op1/ = op2$	$op1 = op1/op2$
%=	Resto	$op1\,\% = op2$	$op1 = op1\,\% op2$
&=	Y	$op1\& = op2$	$op1 = op1\&op2$
$\vert$=	O	$op1\vert = op2$	$op1 = op1\vert op2$
ˆ =	O exclusivo	$op1^\wedge = op2$	$op1 = op1^\wedge op2$
= >>	Desplazamiento	$op1 >>= op2$	$op1 = op1 >> op2$
= <<	Desplazamiento	$op1 <<= op2$	$op1 = op1 << op2$

Nota:

La instrucción $op1+ = op2$ se lee: «el nuevo valor de $op1$ va a ser igual al antiguo valor de $op1$ más $op2$».

Capítulo 3

Estructura de un Programa

Todo programa en pseudocódigo tiene 2 secciones diferenciadas:

- Instrucciones de definición de datos: Definición de los datos de trabajo del programa. Aquí se incluirán todas las constantes y variables de usuario.
- Cuerpo del programa: Grupo de instrucciones que determinan un conjunto de acciones que realiza el programa. A esta sección se le denomina "Algoritmo" o "Cuerpo del programa".

La sintaxis de un programa en pseudo código es la siguiente:

```
PROGRAMA

  DATOS
    CONSTANTES
      <TIPO> <IDENTIF_1> = <VALOR_1>;
      <TIPO> <IDENTIF_2> = <VALOR_2>;
        ....   .........   .   .......
    FINCONSTANTES
    VARIABLES
      <TIPO> <IDENTIF_1>;
      <TIPO> <IDENTIF_2>;
        ....   .........
    FINVARIABLES
  FINDATOS

  ALGORITMO
    INSTRUCCION_1;
    INSTRUCCION_2;
     .............
  FINALGORITMO

FINPROGRAMA
```

Capítulo 4

Instrucciones y Estructuras de Control

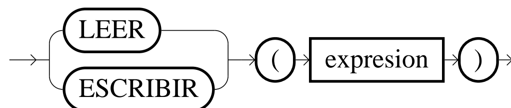
Una instrucción es todo hecho de duración limitada que produce un cambio en el programa.

Las instrucciones se clasifican en diferentes tipos:

4.1. Primitivas de Entrada, Salida y Asignación

4.1.1. Entrada

entrada_salida



LEER(`<Identificador>`);

Lee un valor de la entrada estándar y lo almacena sobre la variable cuyo identificador es el indicado entre paréntesis.

4.1.2. Salida

ESCRIBIR(`<Expresion>`);

Escribe por la salida estándar el valor de la expresión. La expresión puede ser de tipo:

- ENTERO : ESCRIBIR(23)
- REAL : ESCRIBIR(12,4)
- CARACTER : ESCRIBIR('H')
- CADENA DE CARACTERES: ESCRIBIR("Hola")

4.2. Asignación

```
<Identificador> = <Expresion>;
```

Donde:

- <identificador> : Nombre de una variable válida.
- <expresion> : Devuelve el mismo tipo de datos que la variable.

Ejemplo:

```
.....
  ENTERO     numero;
  CARACTER   letra;
......
  numero = ( 23 + 5 );
```

Correcto: La expresión 23 + 5 devuelve (⇒) un resultado 28 que es de tipo ENTERO y el tipo de la variable numero es ENTERO .

```
numero = ( 3 >= 5 );
```

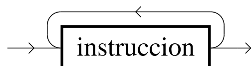
Incorrecto: La expresión 3 = 5 > devuelve un resultado FALSO que es de tipo LOGICO y el tipo de la variable numero es ENTERO . Existiría una incompatibilidad de tipos.

```
letra = 'b';
```

Correcto: La expresión 'b' devuelve un resultado de tipo CARACTER y el tipo de la variable letra es CARACTER .

4.3. Bloques

BLOQUE



Un bloque es un conjunto de instrucciones simples que se ejecutan de manera secuencial. Cuando se representa un bloque todas las instrucciones se sitúan justificadas a la izquierda.

```
instruccion_1;
instruccion_2;
instruccion_3;
instruccion_4;
```

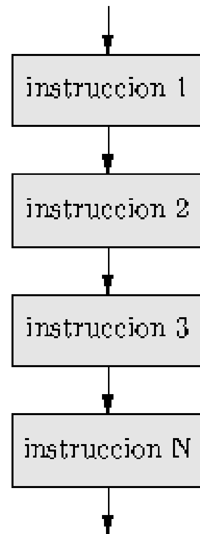


Figura 4.1: Diagrama de un bloque de instrucciones.

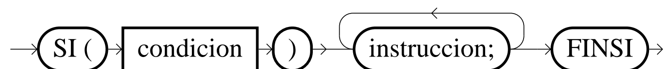
4.4. Alternativas

Una alternativa en un programa representa la ejecución de conjunto de instrucciones en función del resultado de una condición. Desde el punto de vista del programa, una alternativa representa una *bifurcación* en el código del programa.

Existen 3 tipos de alternativas:

4.4.1. Condicional simple

SI




```
SI ( <condicion> )
    Instruccion1;
    Instruccion2;
    ............;
FINSI
```

Donde:

<condicion> : Expresión que devuelve un resultado de tipo lógico: VERDADERO o FALSO .

Funcionamiento:

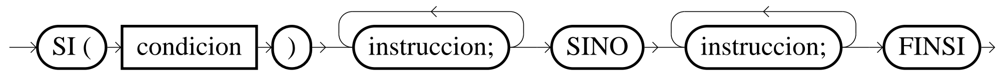
1. Comprueba el valor de la condición.
2. Si el valor de la condición es VERDADERO ⇒Ejecuta el bloque de instrucciones y finaliza.
3. Si el valor de la condición es FALSO ⇒Finaliza.

Ejemplo 1 - SI.

```
.....
SI ( numero == 4 )
   ESCRIBIR("EL NUMERO ES 4");
FINSI
.......
```

4.4.2. Condicional compuesta

SINO




```
SI ( <Condicion> )
   Instruccion1;
   Instruccion2;
   ............;
SINO
   Instruccion1;
   Instruccion2;
   ............;
FINSI
```

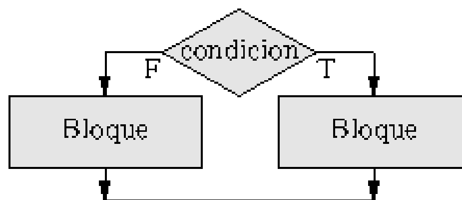


Figura 4.2: Bifurcación de la rama de ejecución en una alternativa.

Funcionamiento:

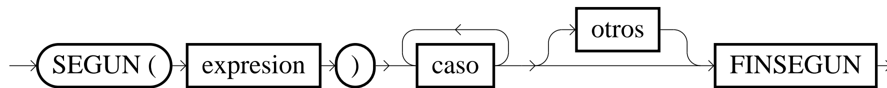
1. Comprueba el valor de la condición.
2. Si el valor de la condición es VERDADERO ⇒ejecuta el bloque de instrucciones 1 y finaliza.
3. Si el valor de la condición es FALSO ⇒Ejecuta el bloque de instrucciones 2 y finaliza.

Ejemplo 2 - SI-SINO.

```
.......
SI ( Numero == 4 )
   ESCRIBIR("EL␣NUMERO␣ES␣4");
SINO
   ESCRIBIR("EL␣NUMERO␣NO␣ES␣EL␣4");
FINSI
.......
```

4.4.3. Condicional múltiple

SEGUN




```
SEGUN( <Expresion> )
  CASO Valor1:  Instruccion1;
                Instruccion2;
                ............;
               FINCASO

  CASO Valor2:  Instruccion1;
                Instruccion2;
                ............;
               FINCASO
 ....  .......  .............
  CASO ValorN:  Instruccion1;
                Instruccion2;
                ............;
               FINCASO

       OTROS:   Instruccion1;
                Instruccion2;
                ............;
               FINCASO
FINSEGUN
```

Donde:

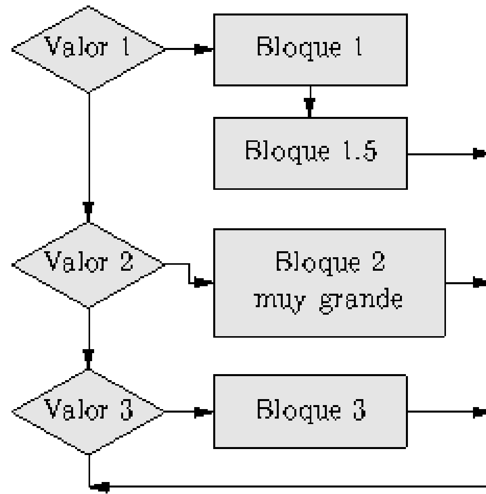


Figura 4.3: Condicional compuesta

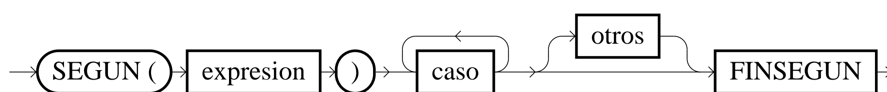
<condicion> : Expresión que devuelve un resultado de tipo lógico: VERDADERO o FALSO . Valor1, Valor2, ValorN : Posibles valores que puede tomar la expresión indicada. También se llaman etiquetas.

Funcionamiento:

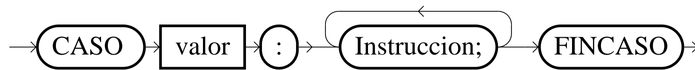
1. Comprueba el valor de la expresión.
2. Busca si el valor de la expresión coincide con alguno de los valores indicados como etiquetas (Valor1, Valor2, ..., ValorN).
3. Si alguna etiqueta-i coincide con el valor de la expresión:
   - Ejecuta el bloque-i asociado a la derecha de la etiqueta.
   - Finaliza SEGUN .
4. Si ninguna etiqueta coincide con el valor de la expresión:
   - Ejecuta el bloque asociado a la derecha de la etiqueta OTROS .
   - Finaliza SEGUN .

Ejemplo 3 - Según.

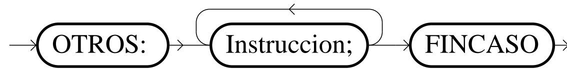
SEGUN



caso



otros



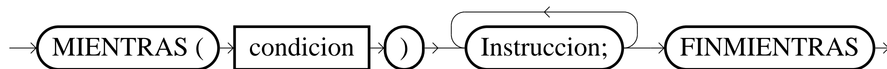

```
      .......
  SEGUN (Numero)
    CASO  1: ESCRIBIR("El numero es 1");
             FINCASO;
    CASO  2: ESCRIBIR("El numero es 2");
             FINCASO;
    CASO  3: ESCRIBIR("El numero es 3");
             FINCASO;
      OTROS: ESCRIBIR("El numero es distinto de 1, 2 y 3");
             FINCASO;
  FINSEGUN
      .......
```

4.5. Iterativas

Una instrucción repetitiva es una instrucción que se ejecuta un número de veces determinado. Este tipo de instrucciones son fundamentales para la creación de programas y también reciben el nombre de "bucles".

4.5.1. Mientras

MIENTRAS




```
MIENTRAS ( <Condicion> )
    Instruccion1;
    Instruccion2;
    ............;
FINMIENTRAS
```

Funcionamiento:

1. Comprueba el valor de la condición.
2. Si la condición devuelve VERDADERO
   - Ejecutar el bloque de instrucciones.
   - Volver al paso 1.

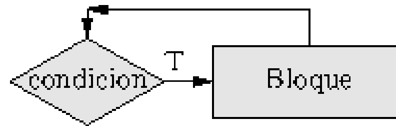


Figura 4.4: Estructura iterativa MIENTRAS.

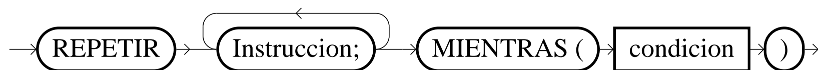
3. Si la condición devuelve FALSO :
   - Finalizar MIENTRAS .

Ejemplo 4 - Mientras.

```
 .......
 Contador = 1;
MIENTRAS (Contador <= 4)
   ESCRIBIR("HOLA");
   Contador = Contador + 1;
FINMIENTRAS;
 ...........
```

4.5.2. Repetir

REPETIR




```
REPETIR
    Instruccion1;
    Instruccion2;
    ............;
MIENTRAS ( <Condicion> )
```

Funcionamiento:

1. Ejecuta el bloque de instrucciones.
2. Comprueba el valor de la condición.
3. Si la condición devuelve VERDADERO
   - Volver al paso 1.
4. Si la condición devuelve FALSO :

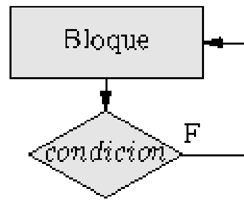
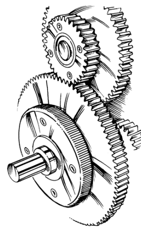


Figura 4.5: Estructura iterativa REPETIR MIENTRAS.

- Finalizar REPETIR .



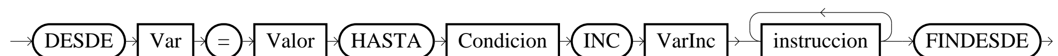
MIENTRAS y REPETIR son casi equivalentes, pero REPETIR se ejecuta al menos una vez. MIENTRAS depende de la condición.

Ejemplo 5 - Repetir

```
 .......
contador = 1;
REPETIR
   ESCRIBIR("HOLA");
   contador = contador + 1;
MIENTRAS( contador <= 4)
...........
```

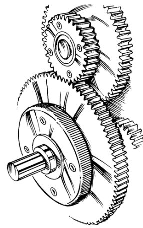
4.5.3. Desde

DESDE




```
DESDE <Var=Valor> HASTA <Condicion> INC(<VarInc>)
    Instruccion1;
    Instruccion2;
```

. . . . ;
FINDESDE

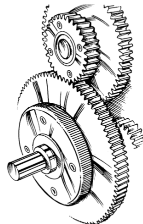


La instrucción DESDE es equivalente a la instrucción MIENTRAS

Donde:

<Var=Valor> : Valor inicial que se le asigna a la variable "var" que se encarga de controlar el bucle. <CondFinal> : Condición que debe de cumplir la variable "var" para continuar iterando en el bucle. <VarInc> : Expresión que indica el incremento o decremento de la variable "var" para cada iteración del bucle.

Funcionamiento:



Cuando se conoce a priori el número de iteraciones, es mejor usar DESDE

1. Ejecutar la Inicialización (Var=Valor);
2. Comprobar el valor de la condición.
3. Si la condición devuelve VERDADERO
   - Ejecutar el bloque de instrucciones.
   - Ejecutar el Incremento/Decremento sobre la variable.

4. Si la condición devuelve FALSO .
   - Finalizar DESDE .

Ejemplo 6 - Bucles ascendentes
Ej1.

```
.....
DESDE Contador=1 HASTA (Contador <= 4) INC(Contador=Contador+1)
   ESCRIBIR("HOLA");
FINDESDE
.....
```

Ej2.

```
.....
DESDE Contador=1 HASTA (Contador <= 4) INC(Contador++)
   ESCRIBIR("HOLA");
FINDESDE
```

Ejemplo 7 - Bucles descendentes
Ej3.

```
.....
DESDE Contador=4 HASTA (Contador >= 1) INC(Contador=Contador-1)
   ESCRIBIR("HOLA");
FINDESDE
.....
```

Ej4.

```
.....
DESDE Contador=4 HASTA (Contador >= 1) INC(Contador--)
   ESCRIBIR("HOLA");
FINDESDE
```

Capítulo 5

Programación modular

Mediante la programación modular es posible crear un conjunto de módulos que interaccionan entre sí y permiten solucionar un problema complejo.

Los tipos de módulos son:

5.1. Funciones

Una función es un módulo que devuelve un valor al programa o al módulo que lo llamó. La sintaxis para la creación de un módulo es:

```
<TIPO> FUNCION(NombreFuncion)
  PARAMETROS
    <TIPO> [VAL/REF] <IDENTIF_1>,
    <TIPO> [VAL/REF] <IDENTIF_2>,
     ....    .......    .........
  FINPARAMETROS

  VARIABLES
    <TIPO> <IDENTIF_1>;
    <TIPO> <IDENTIF_2>;
     ....    .........
  FINVARIABLES

  ALGORITMO
    INSTRUCCION_1;
    INSTRUCCION_2;
     .............
    DEVOLVER (<Expresion>);
  FINALGORITMO

FINFUNCION
```

Donde:

<TIPO> FUNCION(NombreFuncion) :
<TIPO> : Tipo de datos que devuelve la función.
NombreFuncion : Identificador de la función.

<TIPO> [almacenamiento] <IDENTIF1> :
<TIPO> : Tipo de datos al que pertenece al parámetro.
almacenamiento= "VAL" — "REF" : Indica el tipo de parámentro:
VAL : Por valor.
REF : Por referencia.
<IDENTIF1> : Identificador del parámetro.

NOTA 1: No es obligatorio indicar el tipo de parámetro. Si no se indica nada, se suponer VAL por defecto. NOTA 2: Para la definición de varios parámetros, se separan siempre por , menos el último.

DEVOLVER (<Expresion>); Esta instrucción es OBLIGATORIA para cualquier función. La expresión que aparece indicada entre paréntesis será la expresión que devuelva la función al programa o al módulo que la llamó. El tipo de datos que devuelva la instrucción DEVOLVER y el tipo de datos que se indica en la primera línea de la función (<TIPO> FUNCIN(NombreFuncion) ñ) deben de ser iguales. De no ser así habría un error entre el tipo de datos esperado y el devuelto.

Ejemplo 8 - Funciones

```
ENTERO FUNCION(Potencia)
  PARAMETROS
    ENTERO  Base,
    ENTERO  Exponente
  FINPARAMETROS
  VARIABLES
    ENTERO  Res;
    ENTERO  Contador;
  FINVARIABLES
  ALGORITMO
    Res=1;
    DESDE Contador=1 HASTA (Contador<=Exponente) INC(Contador=Contador+1)
      Res = Res * Base;
    FINDESDE
    DEVOLVER (Res);
  FINALGORITMO
FINFUNCION

PROGRAMA
  DATOS
    VARIABLES
```

```
        ENTERO Num1,Num2, Resultado;
      FINVARIABLES
    FINDATOS
    ALGORITMO
      ESCRIBIR("Introduce la base: ");
      LEER(Num1);
      ESCRIBIR("Introduce el exponente: ");
      LEER(Num2);
      Resultado = Potencia(Num1,Num2);
      ESCRIBIR("La potencia vale: ");
      ESCRIBIR(Resultado);
    FINALGORITMO
FINPROGRAMA
```

5.2. Procedimientos

Un procedimiento es un módulo que realiza un conjunto de instrucciones determinadas y que *no devuelve* ningún valor al programa o al módulo que lo llamó. Al no devolver ningún valor, no contienen la instrucción DEVOLVER en el cuerpo del módulo.

```
PROCEDIMIENTO(NombreProcedimiento)

      PARAMETROS
      <TIPO> [VAL/REF] <IDENTIF_1>,
      <TIPO> [VAL/REF] <IDENTIF_2>,
        ....     .......     .........
    FINPARAMETROS

    VARIABLES
      <TIPO> <IDENTIF_1>;
      <TIPO> <IDENTIF_2>;
        ....     .........
    FINVARIABLES

    ALGORITMO
      INSTRUCCION_1;
      INSTRUCCION_2;
       .............
    FINALGORITMO

FINPROCEDIMIENTO
```

Donde:

PROCEDIMIENTO(NombreProcedimiento) :

NombreProcedimiento : Identificador del procedimiento.

<TIPO> [almacenamiento] <IDENTIF1> :
<TIPO> : Tipo de datos al que pertenece al parámetro.
almacenamiento= "VAL" — "REF" : Indica el tipo de parámentro:
VAL : Por valor.
REF : Por referencia.
<IDENTIF1> : Identificador del parámetro.
Ejemplo 9 - Procedimientos

```
PROCEDIMIENTO( DibujaCuadrado )
  PARAMETROS
    ENTERO Lado
  FINPARAMETROS
  VARIABLES
    ENTERO NFila;
    ENTERO NColumna;
  FINVARIABLES
  ALGORITMO
    DESDE NFila=1 HASTA (NFila<=Lado) INC(NFila=NFila+1)
      DESDE NColumna=1 HASTA (NColumna<=Lado) INC(NColumna=NColumna+1)
        SI ( (NFila==1)     OR (NFila==Lado) OR
             (NColumna==1) OR (NColumna==Lado) )
            ESCRIBIR("*");
        SINO
            ESCRIBIR("␣");
        FINSI
      FINDESDE
      ESCRIBIR(SALTO);
    FINPARA
  FINALGORITMO
FINPROCEDIMIENTO

PROGRAMA
  DATOS
    VARIABLES
      ENTERO Numero;
    FINVARIABLES
  FINDATOS
  ALGORITMO
    ESCRIBIR("Introduce␣el␣lado␣del␣cuadrado:␣");
    LEER(Numero);
    DibujaCuadrado(Numero);
  FINALGORITMO
FINPROGRAMA
```

Parte II

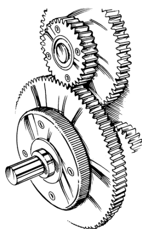
C

Capítulo 6

Introducción

Esta es una guía que te acompañará en tu aprendizaje del lenguaje de programación C. No pretende ser completa, porque sabemos que nada es completo y la experiencia del devenir no tiene sustito en la categorización de las entidades: La vida, es proceso.

Lo que sí pretenden estas páginas es guiarnos por lo fundamental y advertirnos de los riesgos más usuales que acechan en el camino. También aconseja sobre los *modus operandi* y los distintos estilos que los años han sedimentado.



¡Suerte y cuidado con las macros ocultas!

6.1. Los Cinco Mandamientos del Programador

Generalmente el programador sabe resolver el problema que tiene por delante. Pues muchas veces es capaz de resolver el problema a mano: sin ayuda del ordenador. La cuestión no es resolver el problema, sino expresarlo en lenguaje C.

El lenguaje natural tiene una alto nivel de abstracción y, además, se pueden dar muchas cosas por supuestas. Cada una de las palabras del código

máquina/ensamblador expresan muy poco –bajo nivel semántico– y, el orden importa mucho. Un lenguaje de programación de alto nivel tiene un nivel semántico superior cubriendo parte del hueco semántico, el compilador.

Dado que expresar los problemas programáticamente es complicado para los principiantes, presentamos una serie de pasos que simplifican grandemente el problema. Lo único necesario para que funcionen es fe en ellos. Los llamamos: los cinco mandamientos del programador:

1. Comprender el enunciado en todos los casos. (orgullo)
2. Resolver el problema a mano, sin pensar en el ordenador. Conviene hacer un número suficiente de casos. (pereza)
3. Abstraer el problema, sustituyendo los números por conceptos. Es decir, parametrizando. (impaciencia)
4. Escribir en lenguaje esquemático los pasos necesarios para resolver el problema. El pseudocódigo.
5. Comprobar que lo que hemos dicho es lo que queríamos decir. Seguimiento. (soberbia)

6.2. Los componentes del lenguaje

El lenguaje C se compone del preprocesador, el compilador y la librería estándar.

El preprocesador lee el código una vez y hace las sustituciones oportunas antes de que el compilador convierta el fichero fuente en un fichero máquina ejecutable –y cargable por el sistema operativo–. La librería estándar son un conjunto de funciones y constantes que se distribuyen junto al lenguaje C, pero que no forman parte intrínseca del mismo.

Las palabras constituyentes del lenguaje se llaman *palabras clave* y son:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Para compilar el código es se realizan tres análisis:

1. Léxico: Se identifican las palabras y se sustituyen por el tipo sintáctico (token) al que pertenecen.

2. Gramatical: Se comprueba que se ha unido correctamente los token.
3. Semántico: Es el más complicado de llevar a cabo. Indaga sobre el sentido de lo que se quiere decir.

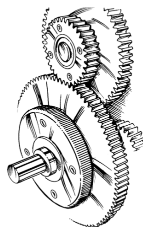
6.3. Los Componentes Sintácticos

Existen seis tokens distintos:

- Palabras clave.
- Identificadores.
- Constantes.
- Cadenas de caracteres.
- Operadores.
- Separadores.

6.3.1. identificadores

Los identificadores son los nombres que se le asigna a variables y funciones. Cada empresa suele tener una guía de estilo indicando la normativa a la hora de elegir identificadores. Sea cual sea la normativa de la empresa es de vital importancia utilizar nombres prólijos, suficientemente descriptivos. Una variable, como norma general, no debe llamarse *n* o *i*, sino *posible_primo* u *operando1*. Evite nombres genéricos como por ejemplo *numero*. Los nombres largos ayudan a leer el código y a pensar con mayor claridad.



Usa nombres descriptivos en los identificadores.

Hay dos maneras extendidas a la hora de juntar palabras en los identificadores: usando el guión bajo, o poniendo la primera letra de la segunda

palabra en mayúsculas –notación de camello–. Tenemos así: *posible_primo* o *posiblePrimo*.

A pesar, que ambas tendencias están bastante enfrentadas, el guión bajo se considera más cortés porque es más fácil su interpretación para los no anglosajones. Habitualmente el código se escribe en inglés para ampliar la usabilidad y mantenibilidad del mismo.

Los identificadores de variables y funciones siempre empiezan con una letra minúscula y los tipos de datos y las clases con una mayúscula.

6.3.2. Cadenas de Caracteres

Las cadenas de caracteres son un tipo especial en C. Cada vez se encuentra una se escriben todos y cada uno de los caracteres y se añade un 0 –expresado como '\0' como carácter–. Por último, se devuelve la dirección de memoria en la que se ha almacendado el primer carácter. Así:

```
"hola"
```

queda

'H'	'o'	'l'	'a'	'\0'

Figura 6.1: Cadena de caracteres en memoria.

Las cadenas de caracteres van entrecomilladas con dobles comillas, aunque es posible:

```
"Esto es " "la misma cadena"
```

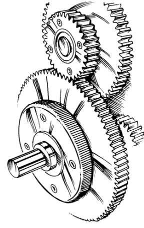
Si se alcanza el final de linea sin cerrar las comillas, el compilador arroja un error, pero como en todos los sistemas *nix se puede indicar que la linea sigue poniendo un *backslash* – \– al final de linea.

```
"Esto es \
   la misma cadena"
```

6.3.3. Comentarios

Los comentarios clarifican el código para los otros programadores – actuales y venideros –. Por ello, no deben ser largos ni farragosos de leer, pero tampoco demasiado escuetos.

Hay comentarios de una línea: empiezan con una barra y terminan al final de la linea. Pero, también hay comentarios de más de una línea. Empiezan por barra asterisco y terminan por asterisco barra.



El código se comenta mientras se programa. No conocemos a nadie que haya comentado el código después de hecho.

```
// Comentario de una linea
/*
  Comentarios de
  varias lineas
*/
```

Los comentarios los elimina el preprocesador junto con los *whitespaces* –espacios, tabuladores y saltos de linea–. El compilador nunca sabe nada de ellos.

Capítulo 7

Datos

7.1. Tipos de datos

Los tipos de datos permitidos para C son:

char Representa valores enteros –$\mathbb{Z}$– entre 0 y 255. Rango 256.

int Representa valores enteros –$\mathbb{Z}$– con 16/32 bits de precisión. Rango: 65 536/4 294 967 296.

float Representa valores reales –$\mathbb{R}$– con 8 bits de exponente, 23 de mantisa y 1 de signo –4 bytes–.

double Representa valores reales –$\mathbb{R}$– con 11 bits de exponente, 52 de mantisa y 1 de signo –8 bytes–.

7.2. Variables y Constantes.

Un dato es toda aquella información relevante que puede ser tratada con posterioridad en un programa. Según el modo de almacenamiento, existen 2 tipos de datos: variables y constantes.

7.2.1. Variables

Declaracion_de_variable

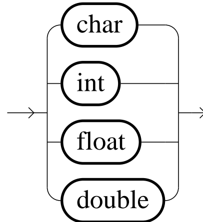
→ TIPO_DE_DATOS → identificador →

```
<TIPO DE DATOS>   <identificador>;
```

Donde:

```
TIPO DE DATOS = "char" | "int" | "float" | "double";
identificador = nombre;
```

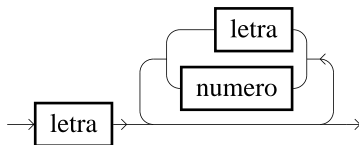
TIPO_DE_DATOS



El identificador es un nombre cualquiera con el que se va a bautizar la variable. En general se recomienda:

```
identificador = minuscula { minuscula | mayuscula | numero };
minuscula     = "a" | "b" | "c" | "d" |
                "e" | "f" | "g" | "h" |
                "i" | "j" | "k" | "l" |
                "m" | "n" | "o" | "p" |
                "q" | "r" | "s" | "t" |
                "u" | "v" | "w" | "x" |
                "y" | "z" | "_" ;
```

identificador



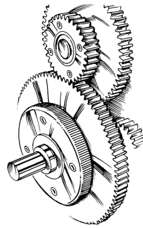
El tipo de datos es fundamental porque le indica al compilador la cantidad de bytes que debe reservar en memoria para leer/almacenar la variable.

Ejemplo:

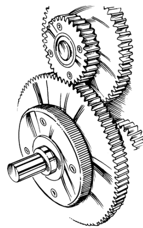
```
int  numero;     # Definicion de la variable
                  # numero de tipo entero.
char letra;  # Definicion de la variable
                  # letra  de tipo caracter.
```

En C no existe el tipo de datos equivalente al valor LOGICO como tal. Para ello se puede utilizar bool , aunque, no obstante, la manera correcta de hacerlo es usar un int . En C se considera verdadero –true – cualquier valor distinto de 0.

Las variables cuentan con un espacio de memoria reservada –memoria estática– antes del programa. Los identificadores son un sinónimo de la di-



Las variables numéricas tienen uso como almacen de datos –lo importante es que guardan un dato–, como contadores –se incrementan generalmente de 1 en 1, pero lo puede hacer de n en n, $n \in \mathbb{R}$– o como acumulador – su valor se incrementa/decrementa en una cantidad–



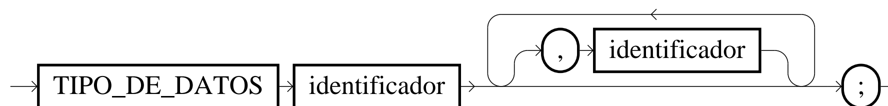
El 0, ni ningún otro valor real –$\mathbb{R}$ – existen con precisión absoluta, por lo que el operador $==$ es inoperativo cuando los reales están involucrados.

rección de memoria donde se encuentra alojada la variable.

Si varias variables comparten el tipo de datos se pueden agrupar los identificadores en una lista separada por comas.

```
<TIPO DE DATOS>   <identificador>[, <identificador>]* ;
```

Declaracion_de_varias_variables



El cualificador const congela el valor de una variable.

Así que de manera general podemos declarar variables.

```
["const"] <TIPO DE DATOS>   <identificador>;
```

Declaracion_de_constvar

Listas

Es posible tener una lista de variables agrupadas y acceder a ellas con un índice.

```
<TIPO DE DATOS>   <identificador>[<cantidad>];
```

Donde *cantidad* indica cuantas variables van a ser creadas. En la tabla de variables se anota la dirección de memoria donde se ha hecho la reserva de memoria y la cantidad de bytes que ocupa dicha reserva.

```
int par[] = {2, 4, 6, 8};
```

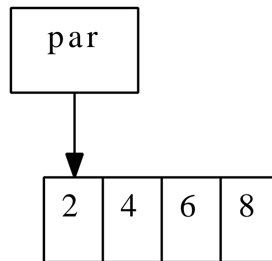
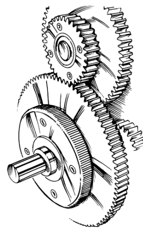


Figura 7.1: Una lista de pares.

Es posible dejar la cantidad en blanco si hay una inicialización, ya que el preprocesador contará los elementos por nosotros y rellenará la cantidad.



El identificador es una variable distinta de la reserva que contiene la dirección de memoria del primer elemento.

Se pueden crear listas de mas de una dimensión. Matrices.

```
int tablero[8][8];
```

El primer índice suele considerarse como filas, y el segundo como columnas.

Nótese que habiendo 8 elementos, estos estarán numerados de 0 a 7 ambos incluidos. Para acceder a los elementos de un Array usamos su índice.

```
tablero[2][4] = 5;
```

En principio, no se hacen comprobaciones de rango, con lo cual es posible escribir fuera de la matriz o lista. Acto seguido disfrutaremos de los catastróficos efectos subsiguientes.

Punteros

Un puntero es una variable que contiene la dirección de memoria de otra variable. Como las direcciones de memoria cambian no nos referimos a la dirección como tal, sino que decimos que el puntero *apunta* a la variable.

Sea:

```
int area = 2;
int *p;
...
p = &area;
```

Decimos que *p* apunta a *area* y lo representamos:

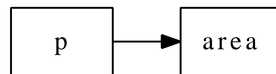


Figura 7.2: p apunta a area.

El operador & se llama operador de dirección, y se lee *la dirección de.* Su contrapartida es el operador *, el operador de indirección, que se lee *el contenido de lo que apunta.* Así, decimos que *p* contiene la dirección de *area* y **p* vale 2.

Los punteros y las listas son tremendamente parecidos. La mayor diferencia estriba en que estas recuerden el número de bytes de su reserva, mientras que los primeros no. Pero, su uso es equivalente en muchos casos.

Véase, por ejemplo:

```
char *p   = "Buenos dias";
char l[]  = "Hola";
```

En la primera declaración se encuentra una cadena de caracteres, que se almacena dentro del programa, y se devuelve la dirección de memoria donde se ha almacenado. Esta dirección se asigna a la variable *p*, que es un puntero a carácter.

Pero, sizeof(p) $\Rightarrow$1, y sizeof(l) $\Rightarrow$5 –contando el '\0' del final–.

Nótese que el tipo de datos es *char ** y que a pesar del espacio, todo es el tipo de la variable. No se debe confundir este asterisco, que forma parte indisoluble de *char* con el operador de indirección. Es totalmente necesario tipar los punteros. Ya que cuando accedemos a una dirección de memoria

–1000 por ejemplo–, si vamos a leer un *char*, leemos un byte. Si vamos a leer un *int*, entonces 2/4 –dependiendo del sistema operativo– bytes.

Para recordar sólo la dirección, sin más, usamos el tipo *void **. Se puede tener un puntero a un puntero a *char*. Su tipo: *char ***.

Cuando se declara un puntero, éste puede contener basura informática, y apuntar a cualquier parte de la RAM. En ese caso se dice que el puntero está *salvaje*. Para evitar esta desagradable situación, todos los punteros deben ser *inicializados* antes de usarse.

7.2.2. Constantes

En C encontramos constantes numéricas –enteras, punto flotante–, tipo carácter, enumeración y constantes simbólicas.

Constantes numéricas

Las constantes numericas son números que están albergadas dentro del código, junto con las instrucciones.

Constantes enteras decimales.

Ejemplos:

```
23485          // constante tipo int
45815          // constante tipo long (es mayor que 32767)
243u           // constante tipo unsigned int
243U           // constante tipo unsigned int
739l           // constante tipo long
739L           // constante tipo long
583ul          // constante tipo unsigned long
583UL          // constante tipo unsigned long
```

Constantes enteras octales. Empiezan por 0 y tienen dígitos entre 0 y 7 [0-7].

```
011  // 9 en decimal
```

Constantes enteras hexadecimales. Empiezan por 0x y tienen dígitos [0-9A-Fa-f].

```
0x1A  // 26 en decimal
0x23  // 35 en decimal
0x1a  // 26 en decimal
```

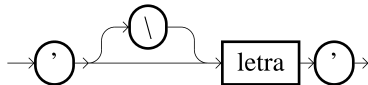
Constantes de tipo Carácter

```
"'" "\"?letra "'"
```

Secuencia	Valor	Símbolo	Descripción
\a	0x07	BEL	Sonido audible (BELL)
\b	0x08	BS	Retroceso (Back Space)
\n	0x0C	FF	De avance (Form Feed)
\f	0x0A	LF	Salto de línea (Line Feed)
\r	0x0D	CR	Retorno de carro (Carriage Return)
\t	0x09	HT	Tabulación Horizontal (Horizontal Tab)
\v	0x0B	VT	Tabulación Vertical (Vertical Tab)
\\	0x5C	\	Barra invertida (backslash)
\’	0x27	’	Apóstrofe (apostrophe)
\”	0x22	”	Comillas (quotes)
\?	0x3F	?	Interrogación (question mark)
\O	xxxx	xxxx	O: Hasta tres dígitos octales
\xH	xxxx	xxxx	H: Cualquier número hexadecimal menor de 256
\XH	xxxx	xxxx	H: Cualquier número hexadecimal menor de 256

Cuadro 7.1: Secuencias de escape.

constante_caracter



Sustitución:

El código ASCII es un código de 7 bits – posteriormente extendido a 8– donde a cada número se le hace corresponder una grafía. Las 32 primeras entradas están destinadas al control y no a la representación. El preprocesador sustituye la letra por el número que ocupa dicha letra dentro del código ascii.

Si la constante incluye la \, hablamos de secuencias de escape. Las secuencias de escape especifican el número del carácter en octal.

```
'\60'  // Es el numero 48
       // en decimal: que es
       // el codigo del '0'.
```

También se pueden expresar en hexadecimal.

```
'\x30'  // Es el numero 48
        // en decimal: que es
        // el codigo del '0'.
```

Aunque en general si estos códigos tienen asignada una letra, es esto lo que se suele emplear. A modo de resumen presentamos el cuadro 7.1.

Sea, por ejemplo, el fichero de texto `user$ saludo.txt`.

```
cat -> saludo.txt
```

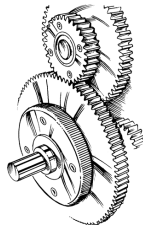
```
Hola
hola<ctrl-d>
```

Cuando hacemos un volcado de octetos del archivo:

```
od -c saludo.txt
0000000   H   o   l   a       \n   h   o   l   a  \n
0000013

 od -b saludo.txt
0000000 110 157 154 141 040 012 150 157 154 141 012
0000013
```

que claramente nos interesa ver en hexadecimal.

```
 od -t x1 saludo.txt
0000000 48 6f 6c 61 20 0a 68 6f 6c 61 0a
0000013
```



Las minúsculas son mayores que las mayúsculas. Nótese que para cualquier par de letras su distancia es 0x20 — 'h' - 'H' = 0x20 —

Constantes simbólicas

```
#define <identificador> <valor de sustitucion>
```

Declaracion_de_constante_simbolica

→ (#define) → [identificador] → [valor_de_sustitucion] →

Son nombres asignados a constantes. Aportan claridad al código porque nos hablan del concepto, no del número. También aportan mantenibilidad, porque los cambios en el valor sólo se hacen en la definición.

Ejemplo:

```
#define PI 3.14159
```

El valor de sustitución es todo lo que se ponga desde el identificador hasta el final de la línea. Por lo tanto no es correcto:

```
#define PI 3.14159 // Es el numero PI
```

Sin embargo, sí es correcto:

```
#define PI 3.14159 /* Es el numero PI */
```

El encargado de las sustituciones es el preprocesador que sólo lee el código una vez.

Sería correcto:

```
#define l 2.5
#define area l*l
```

Pero no:

```
#define area l*l
#define l 2.5
```

ya que la definición de area, incluye el valor desconocido l

Una vez que una constante simbólica ha sido definida queda registrada en una lista, aunque la constante no tega valor.

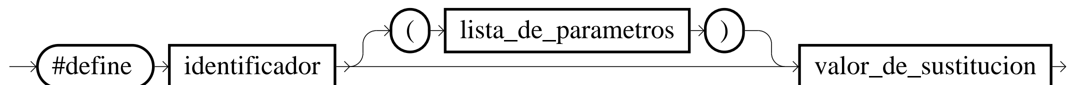
```
#define AREA
```

También es posible sustituir un nombre por un trozo de código, en cuyo caso hablamos de *macros*, y al proceso de sustitución se le llama *expansión* de la macro.

Así llegamos a la siguiente definición:

```
#define <identificador>[(<lista de parametros>)] <valor de sustitucion>
```

Declaracion_de_macros



Ejemplo:

```
#define AREA l*l

void main(){
  int l = 2;
  int area;

  area = AREA;
}
```

Las operaciones en las macros suelen ir encerradas entre paréntesis. Nótese lo incorrecto de esta operación:

```
#define PERIMETRO l + l + l + l

void main(){
  int l = 2;
  int doble_perimetro;

  doble_perimetro = 2 * PERIMETRO;
}
```

En esta operación el resultado es $2 \times l + l + l + l = 2 \times 2 + 2 + 2 + 2 = 4 + 2 + 2 + 2 = 10$ que difiere de los 16 esperados.

En el caso de que un dato difiera dependiendo de la parte del código, hablamos de parámetros.

```
#define MIN(a,b) ((a)>(b)?(b):(a))
```

Para más información ver preprocesador de c en la wikipedia.

Constantes enumeración

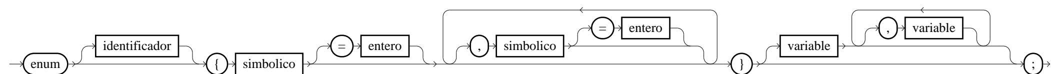
A veces es necesario definir constantes con valores correlativos. Para ello, siempre es posible hacer:

```
#define LUNES     0
#define MARTES    1
#define MIERCOLES 2
#define JUEVES    3
#define VIERNES   4
#define SABADO    5
#define DOMINGO   6
```

No obstante, contamos con *enum*

```
enum [<identificador>] {  <simbolico>[=<entero>]
                           [,<simbolico>[=<entero>]]+ }
                          [variable[, variable]+] ;
```

Declaracion_de_constante_enum



Valga como ejemplo:

```
enum laboral {lunes=1, martes, miercoles, jueves, viernes};
```

En donde se definen las constantes lunes, martes, miercoles, jueves y viernes con valores correlativos 1,2,3,4,5. Si se omitiese el `=1` la numeración comenzaría en 0.

También es posible hacer:

```
enum laboral {lunes, martes, miercoles=5, jueves, viernes};
```

Los valores de las constantes, serán: 0,1,5,6,7.

Además de las constantes, estamos definiendo el tipo de datos laboral, y podemos declarar variables que contienen lunes, martes, miercoles, jueves y viernes. Hay dos maneras de hacerlo: en la definición, o en una declaración a parte.

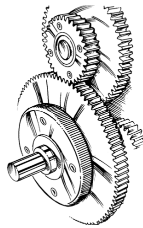
```
enum laboral {lunes, martes, mier, jueves, viernes} dia_libre;
```

o

```
enum laboral {lunes, martes, mier, jueves, viernes};
.
.
.
enum laboral dia_libre;
```

Y se le asigna un valor a la variable:

```
dia_libre = mier;
```



Las variables y las constantes tipo enum no valen como índices en las listas.

Capítulo 8

Operadores

Los operadores determinan las diferentes operaciones que se pueden realizar son los operandos. Los operadores se clasifican en:

8.1. Operadores aritméticos

Realizan una operación aritmética, los operandos siempre son de tipo numérico (ENTERO o REAL) y devuelven un resultado numérico (ENTERO o REAL).

Operador	Nombre	Lectura	Modo de Uso
+	Suma	más	$operando1 + operando2 \Rightarrow suma$
-	Resta	menos	$operando1 - operando2 \Rightarrow resta$
*	Multiplicación	por	$operando1 * operando2 \Rightarrow multiplicacion$
/	División	entre	$operando1 / operando2 \Rightarrow division$
%	Resto	resto	$operando1 \% operando2 \Rightarrow resto$

Ejemplo:

```
23   +  2   => 25
 3   - 12   => -9
 7,5 *  2   => 15
12,4 /  2   => 6,2
 5   %  2   => 1
```

8.2. Operadores Relacionales

Comprueban la relación existente entre dos operandos. Los operandos pueden ser de cualquier tipo siempre que ambos sean del mismo tipo. Devuelven un resultado lógico (true ó false).

Operador	Nombre	Lectura	Modo de Uso
==	igualdad	es igual a	$operando1 == operando2$
! =	desigualdad	es distinto de	$operando1 != operando2$
<	menor	es menor que	$operando1 < operando2$
<=	menor o igual	es menor o igual que	$operando1 <= operando2$
>	mayor	es mayor que	$operando1 > operando2$
<=	mayor o igual	es mayor o igual que	$operando1 >= operando2$

Ejemplo:

```
23 ==  4 => false
 7 !=  8 => true
34 <  12 => false
12 <= 12 => true
 4 >  -2 => true
-4 >=  1 => false
```

8.3. Operadores Lógicos

Realizan una operación lógica. Los operandos deben ser de tipo lógico (true ó false) y devuelven un resultado lógico.

Operador	Nombre	Lectura	Modo de Uso
NOT	Negación	no	NOT operando
AND	Y Lógico	y	operando1 AND operando2
OR	O Lógico	o	operando1 OR operando2

Ejemplo:

```
NOT true          => false
NOT false         =>  true
true  AND  true   =>  true
true  AND false   => false
false AND  true   => false
false AND false   => false
true   OR  true   =>  true
true   OR  false  =>  true
false  OR  true   =>  true
false  OR false   => false
```

8.4. Operadores a Nivel de Bits

Los operadores a nivel de bits toman cada uno de los bits de una cifra, cuando está expresada en binario, como si fuera un valor lógico.

Operador	Nombre	Lectura	Modo de Uso
&	and	and	$operando1 \& operando2$
$\vert$	or	or	$operando1 \vert operando2$
$\sim$	not	not	$operando1 \sim operando2$
^	exclusive or	x-or	$operando1 \wedge operando2$
>>	desplazamiento	desplazado operando2 posiciones a la der.	$operando1 >> operando2$
<<	desplazamiento	desplazado operando2 posiciones a la izq.	$operando1 << operando2$

Ejemplo:

6 & 5 => 4
˜ 5 => 2
6 $\vert$ 5 => 7
6 ˆ 5 => 3
23 >> 2 => 5
4 << 1 => 8

8.5. Operadores taquigráficos

Los operadores taquigráficos no son esencialmente distintos de los operadores ya vistos, sino una manera abreviada de usarlos como acumulador.

Operador	Nombre	Modo de Uso	equivalencia
+=	Suma	$op1+=op2$	$op1 = op1 + op2$
-=	Resta	$op1-=op2$	$op1 = op1 - op2$
*=	Multiplicación	$op1*=op2$	$op1 = op1 * op2$
/=	División	$op1/=op2$	$op1 = op1/op2$
%=	Resto	$op1\%=op2$	$op1 = op1\%op2$
&=	Y	$op1\&=op2$	$op1 = op1\&op2$
$\vert$=	O	$op1\vert=op2$	$op1 = op1\vert op2$
ˆ =	O exclusivo	$op1^\wedge=op2$	$op1 = op1^\wedge op2$
$\gg$=	Desplazamiento	$op1 \gg= op2$	$op1 = op1 \gg op2$
$\ll$=	Desplazamiento	$op1 \ll= op2$	$op1 = op1 \ll op2$

Nota:

La instrucción $op1+=op2$ se lee: «el nuevo valor de $op1$ va a ser igual al antiguo valor de $op1$ más $op2$».

Capítulo 9

Estructura de un Programa

Todo programa en pseudocódigo tiene 2 secciones diferenciadas:

- Instrucciones de definición de datos: Definición de los datos de trabajo del programa. Aquí se incluirán todas las constantes y variables de usuario.
- Cuerpo del programa: Grupo de instrucciones que determinan un conjunto de acciones que realiza el programa. A esta sección se le denomina "Algoritmo" o "Cuerpo del programa".

La sintaxis de un programa en pseudo código es la siguiente:

```
PROGRAMA

  DATOS
    CONSTANTES
      <TIPO> <IDENTIF_1> = <VALOR_1>;
      <TIPO> <IDENTIF_2> = <VALOR_2>;
        ....    .........   .   .......
    FINCONSTANTES
    VARIABLES
      <TIPO> <IDENTIF_1>;
      <TIPO> <IDENTIF_2>;
        ....    .........
    FINVARIABLES
  FINDATOS

  ALGORITMO
    INSTRUCCION_1;
    INSTRUCCION_2;
     .............
  FINALGORITMO

FINPROGRAMA
```

Capítulo 10

Instrucciones y Estructuras de Control

10.1. Primitivas de Entrada, Salida y Asignación

10.2. Asignación

```
<Identificador> = <Expresion>;
```

Donde:

- <identificador> : Nombre de una variable válida.
- <expresion> : Devuelve el mismo tipo de datos que la variable.

Ejemplo:

```
.....
   int  numero;
   char letra;
......
   numero =  23 + 5  ;
```

Correcto: La expresión 23 + 5 devuelve (⇒) un resultado 28 que es de tipo int y el tipo de la variable numero es int .

```
numero = ( 3 >= 5 );
```

Incorrecto: La expresión 3 = 5 > devuelve un resultado false que es de tipo bool y el tipo de la variable numero es int . Existiría una incompatibilidad de tipos.

```
letra  = 'b';
```

Correcto: La expresión 'b' devuelve un resultado de tipo char y el tipo de la variable letra es char .

10.2.1. Salida

Las instrucciones de entrada-salida no están contempladas en el lenguaje C como tal, sino en las librerías estándar de C.

A pesar de que la entrada y salida puede refererirse a un dispositivo en concreto, se prefiere trabajar con el concepto de flujo –*stream* –. Un flujo lo podemos modelar como un canal con dos puntos terminales en el que se introducen bytes por un extremo y se recogen por otro. Por cuestiones de optimización los bytes no se suelen enviar nada más ser introducidos, sino que quedan almacenados en un parking –*buffer* – hasta que se juntan un número suficiente de ellos, y son enviados. Al vaciado de un *buffer* se lo conoce como *flush* .

Los flujos por defectos son stdin – la entrada estándar– y stdout –salida estándar–, que suelen hacer referencia al teclado y la pantalla, pero es perfectamente posible redirigirlos hacia la red o hacia un fichero.

printf

Libreria: stdio (standard input output)

Es por tanto necesario incluir esa librería en el prólogo del programa.

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

```
 printf( cadena, ...);
```

La función printf requiere al menos una cadena de caracteres y soporta un número variable de argumentos.

Ejemplos:

```
 printf("Hola");
```

```
 printf("Hola " "Mundo");
```

```
 printf("Hola \
 Mundo");
```

Las funciones arriba mostradas sólo tienen un parámetro. No obstante, es posible es posible sustituir dentro de la cadena algún valor, usando los especificadores de formato.

```
 printf("lado = %i, area = %i", 24, 24 * 24);
```

putX

```
int fputc(int c, FILE *stream);

int fputs(const char *s, FILE *stream);
```

```c
int putc(int c, FILE *stream);

int putchar(int c);

int puts(const char *s);
```

- fputc() escribe c, moldeado a unsigned char, al stream.
- fputs() escribe la cadena al stream, sin el '\0'.
- putc() equivalente a fputc() pero como macro.
- putchar(c); equivalente a putc(c,stdout).
- puts() escribe la cadena y un salto de linea en stdout.

10.2.2. Entrada

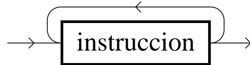
getX

```c
int fgetc(FILE *stream);

char *fgets(char *s, int size, FILE *stream);

int getc(FILE *stream);

int getchar(void);

char *gets(char *s);

int ungetc(int c, FILE *stream);
```

- fgetc() lee el siguiente carácter del stream y lo devuelve como un unsigned char moldeado a int , or *EOF* si se llega al final de fichero o hay un error.
- getc() es equivalente a fgetc(), pero es una macro.
- getchar() es equivalente a getc(stdin).
- gets() lee una linea de stdin y la carga en el *buffer* apuntado por s hasta nueva línea o *EOF* , y pone '\0' como marca de final de cadena. No se comprueba el desbordamiento de *buffer*
- fgets() lee varias lineas y pone al final un '\0'. Se especifica el número de caracteres a leer.
- ungetc() Devuelve c al buffer para que pueda ser releido. Sólo se garantiza poder empujar un caracter de vuelta.

10.3. Bloques

BLOQUE



Un bloque es un conjunto de instrucciones simples que se ejecutan de manera secuencial. La agrupación de instrucciones se realiza empleando corchetes. Los bloques se indentan un número fijo de espacios – no tabulador –.

```
{
    instruccion_1;
    instruccion_2;
    instruccion_3;
    instruccion_4;
}
```

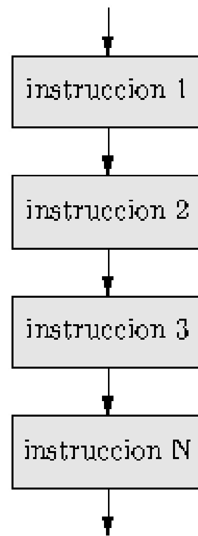


Figura 10.1: Diagrama de un bloque de instrucciones.

10.4. Alternativas

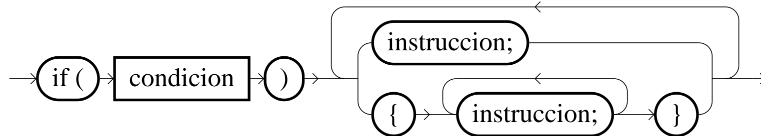
Una alternativa en un programa representa la ejecución de conjunto de instrucciones en función del resultado de una condición. Desde el punto de

vista del programa, una alternativa representa una *bifurcación* en el código del programa.

Existen 3 tipos de alternativas:

10.4.1. Condicional simple

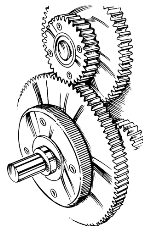
if




```
if ( <condicion> ){
    Instruccion1;
    Instruccion2;
    ............;
}
```

Donde:

<condicion> : Expresión que devuelve un resultado de tipo lógico: VERDADERO o FALSO .



Los bloques de una sola instrucción no van nunca entre corchetes. Igual que en matemáticas los paréntesis no se ponen si no son necesarios, en programación pasa lo mismo.

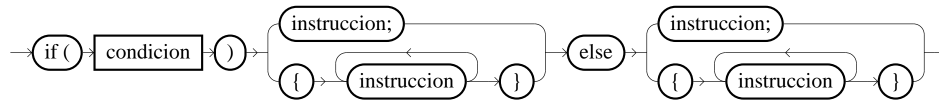
Funcionamiento:

1. Comprueba el valor de la condición.
2. Si el valor de la condición es true ⇒Ejecuta el bloque de instrucciones y finaliza.
3. Si el valor de la condición es false ⇒Finaliza.

Ejemplo 1 - if.

```
.....
if ( numero == 4 )
    printf("El numero es 4");
.......
```

10.4.2. Condicional compuesta




```
if ( <Condicion> ){
   Instruccion1;
   Instruccion2;
   ............;
} else {
   Instruccion1;
   Instruccion2;
   ............;
}
```

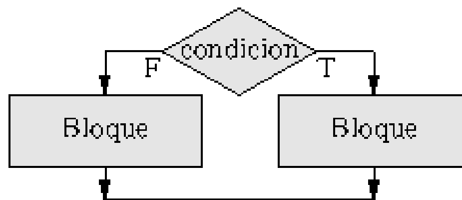


Figura 10.2: Bifurcación de la rama de ejecución en una alternativa.

Funcionamiento:

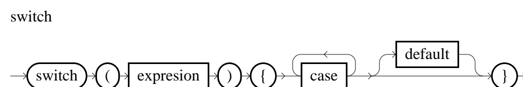
1. Comprueba el valor de la condición.
2. Si el valor de la condición es true ⇒ejecuta el bloque de instrucciones 1 y finaliza.
3. Si el valor de la condición es false ⇒Ejecuta el bloque de instrucciones 2 y finaliza.

Ejemplo 2 - if - else.

```
.......
if ( numero == 4 )
   printf("EL NUMERO ES 4");
else
   printf("EL NUMERO NO ES EL 4");
.......
```

10.4.3. Condicional múltiple

switch




```
 switch( <Expresion> ){
  case Valor1: Instruccion1;
               Instruccion2;
               ............;
               break;

  case Valor2: Instruccion1;
               Instruccion2;
               ............;
               break;
  .... ....... ............
  case ValorN: Instruccion1;
               Instruccion2;
               ............;
               break;

      default: Instruccion1;
               Instruccion2;
               ............;
               break;
```

FINSEGUN

Donde:

<condicion> : Expresión que devuelve un resultado de tipo lógico: true o false . Valor1, Valor2, ValorN : Posibles valores que puede tomar la expresión indicada. También se llaman etiquetas.

Funcionamiento:

1. Comprueba el valor de la expresión.
2. Busca si el valor de la expresión coincide con alguno de los valores indicados como etiquetas (Valor1, Valor2, ..., ValorN).
3. Si alguna etiqueta-i coincide con el valor de la expresión:

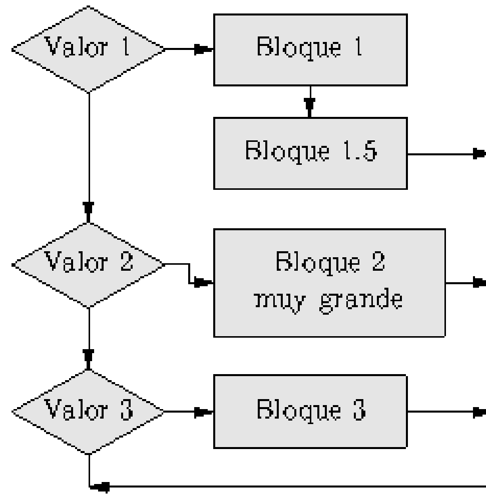


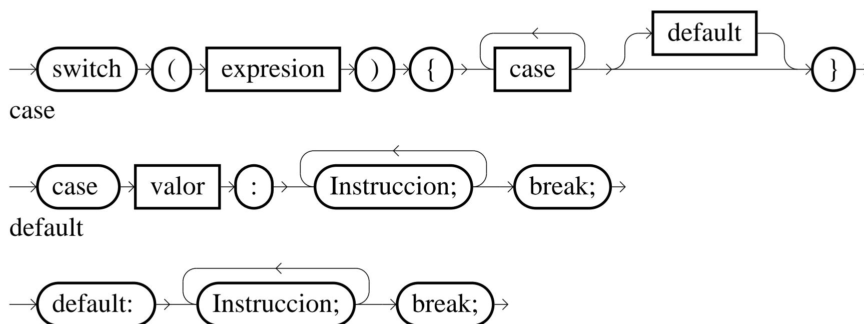
Figura 10.3: Condicional compuesta

- Ejecuta el bloque-i asociado a la derecha de la etiqueta.
- Finaliza switch .

4. Si ninguna etiqueta coincide con el valor de la expresión:
   - Ejecuta el bloque asociado a la derecha de la etiqueta default .
   - Finaliza switch .

Para mayor claridad, ampliamos el diagrama de carril superior.

switch



Ejemplo 3 - Switch.

```
.......
switch (numero){
```

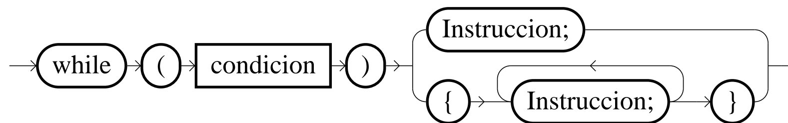
```
    case    1:  printf("El numero es 1");
                break;
    case    2:  printf("El numero es 2");
                break;
    case    3:  printf("El numero es 3");
                FINCASO;
    default:    printf("El numero es distinto de 1, 2 y 3");
                break;
}
      .......
```

10.5. Iterativas

Una instrucción repetitiva es una instrucción que se ejecuta un número de veces determinado. Este tipo de instrucciones son fundamentales para la creación de programas y también reciben el nombre de "bucles".

10.5.1. while

while




```
while ( <Condicion> ){
    Instruccion1;
    Instruccion2;
    ............;
}
```

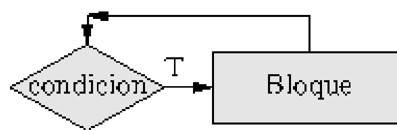


Figura 10.4: Estructura iterativa while.

Funcionamiento:

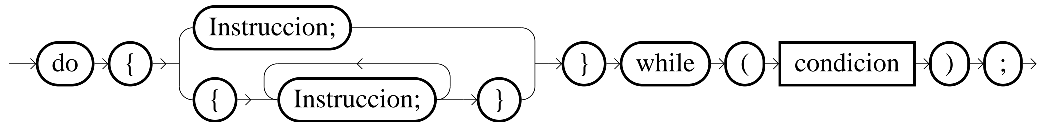
1. Comprueba el valor de la condición.
2. Si la condición devuelve true
   - Ejecutar el bloque de instrucciones.
   - Volver al paso 1.
3. Si la condición devuelve false :
   - Finalizar while .

Ejemplo 4 - While.

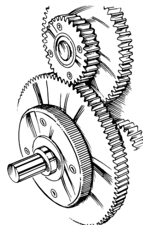
```
.......
contador = 1;
while (Contador <= 4){
  printf("HOLA");
  contador = contador + 1;
}
...........
```

10.5.2. do while

do_while




```
do{
   Instruccion1;
   Instruccion2;
   ............;
}while( <condicion> );
```



Nótese el punto y coma que cierra la estructura.

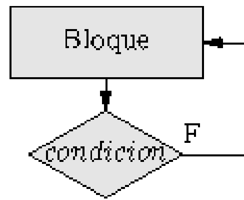
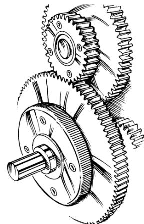


Figura 10.5: Estructura iterativa do - while.

Funcionamiento:

1. Ejecuta el bloque de instrucciones.
2. Comprueba el valor de la condición.
3. Si la condición devuelve true
   - Volver al paso 1.
4. Si la condición devuelve false :
   - Finalizar .



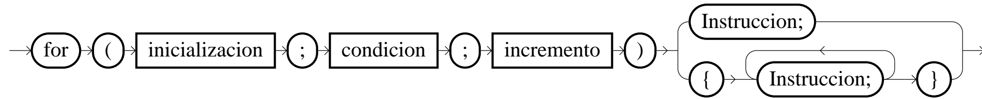
while y do-while son casi equivalentes, pero do-while se ejecuta al menos una vez. while depende de la condición.

Ejemplo 5 - do-while

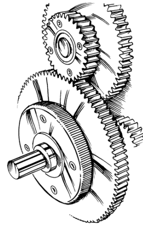
```
 .......
contador = 1;
do{
  printf("HOLA");
  contador = contador + 1;
}while( contador <= 4);
...........
```

10.5.3. for

for




```
for(<inicializacion>; <Condicion>; <Incremento>){
    Instruccion1;
    Instruccion2;
    ............;
}
```



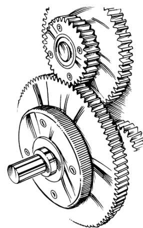
La instrucción for es equivalente a la instrucción while

Donde:

inicializacion , condicion e incremento es cualquier bloque válido en C.

No obstante, la inicialización se suele emplear para darle un valor inicial a la variable al comienzo del bucle. La condicion casi obligatoriamente debe marcar la condición de parada, y el incremento se utiliza para cambiar el valor de la variable de control de la condición.

Funcionamiento:



Cuando se conoce a priori el número de iteraciones, se prefiere for sobre while .

1. Ejecutar la Inicialización (Var=Valor);
2. Comprobar el valor de la condición.
3. Si la condición devuelve true
   - Ejecutar el bloque de instrucciones.
   - Ejecutar el Incremento/Decremento sobre la variable.
4. Si la condición devuelve false .
   - Finalizar for .

Ejemplo 6 - Bucles ascendentes
Ej1.

```
.....
  for(contador=1; contador<=4; contador++)
     printf("HOLA");
 .....
```

Ej2.

```
.....
  for(contador=0; contador<4; contador++)
     printf("HOLA");
 .....
```

El Ejemplo 1 y el Ejemplo 2 son equivalentes y se prefiere, con diferencia la segunda forma sobre la primera. Los programadores, salvo motivos de fuerza mayor comienzan a contar en 0.
Ejemplo 7 - Bucles descendentes
Ej3.

```
.....
  for(contador=4; contador>0; contador--)
     printf("HOLA");
 .....
```

Ej4.

```
.....
  for(contador=3; contador>=0; contador--)
     printf("HOLA");
 .....
```

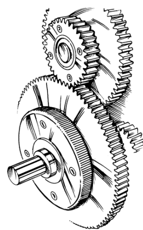
Aunque ambos dos bucles imprimen cuatro veces HOLA, no son equivalentes. En el Ejemplo 3, contador toma los valores $contador = 4, 3, 2, 1$, en el ejemplo 4 $contador = 3, 2, 1, 0$.

Capítulo 11

Programación modular

Mediante la programación modular es posible crear un conjunto de módulos que interaccionan entre sí y permiten solucionar un problema complejo. Cuando agrupamos código bajo un nombre, estamos creando un concepto de mayor complejidad y abstracción –que no más impreciso–. Esto nos permite, además, cambiar la estrategia de diseño del programa. Ya no nos debemos preguntar *cómo* se resuelve el problema, sino *qué* es lo que hay que hacer. Se va diviendo el problema así de los más general, a lo más particular. Esta estrategia recibe el nombre de *análisis descendente* o *análisis top-down*.

En C, a parte de las macros previamente citadas, sólo existen funciones. Los procedimientos son funciones que devuelven nada –void–.



Es distinto devolver nada –función–, que no devolver nada –procedimiento–.

11.1. Funciones

Una función C consta de 5 partes principales: declaración, definición, llamada, parámetros y valor de retorno.

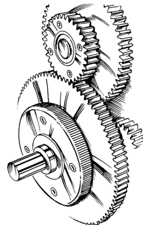
11.1.1. Declaración

La declaración de la función consiste –como en toda declaración– en asociar un identificador con las cantidades y posiciones de memoria relacionadas con su manejo. La primera vez que aparece el nombre de una función es su declaración y, esto es válido sea lo que sea: definición, llamada, etc.

Antigüamente se hacía la declaración explícita de las funciones usando su firma o prototipo al principio del programa.

```
void imprimir(const char*);
double producto(double, double);
```

Hoy en día se usa la propia definición como declaración.



Cuando se usa la definición como declaración, dentro de la definición no se puede hacer llamadas a funciones que todavía no han sido definidas.

11.1.2. Definición

La definición consiste en asociar el nombre de la función –identificador– con el código que la compone.

```
<TIPO> <identificador>(<TIPO> <IDENTIF_1>, ...){
  // Declaracion de variables.
    <TIPO> <IDENTIF_1>;
    <TIPO> <IDENTIF_2>;

  // Algoritmo propiamente dicho
    INSTRUCCION_1;
    INSTRUCCION_2;
    .............
    [return <Expresion>;]
}
```

El primer TIPO que aparece es el tipo de datos del valor de retorno. Si no se ha devolver nada, póngase «void». Después del identificador, o nombre

de la función aparecen los *parámetros formales* que son variables que se rellenan con un valor inicial distinto en cada llamada. El valor que reciben inicialmente se conoce como *parámetro actual*.

En C la declaración de variables locales se hace obligatoriamente antes de empezar a escribir el algoritmo. En C++ se puede hacer en cualquier parte del código –y desde aquí se recomienda declarar las variables justo cuando se necesiten–.

Cuando la función devuelve void return se puede omitir y, si se pone, no va acompañado de ninguna expresión.

C soporta funciones con un número varible de argumentos –como por ejemplo printf–, aunque no son demasiado cómodas de definir.

Ejemplo 8 - Funciones

```
/* Funcion que devuelve el valor
 * absoluto de un numero.
 */
double absoluto(double posible_negativo){
  if (posible_negativo < 0)
    return -posible_negativo;
  return posible_negativo;
}

/* Funcion que potencia un par de numeros */
double potencia(double base, int exponente){
  double resultado = 1;
  for(int i=0; i<absoluto(exponente); i++)
    resultado *= base;
  if (exponente<0)
    resultado = 1 / resultado;
  return resultado;
}
```

11.1.3. Llamada

La llamada consiste en la ejecución del código que define la función. Cuando se ejecuta la llamada se pueden suministrar valores –*parámetros actuales*– que viajarán hasta el código a ejecutar.

Ejemplo 9.

```
...

void main(){
  printf("2 elevado a 3 = %lf",
```

```
        potencia(2., 3) );
}
```

Los *parámetros actuales* son 2 y 3, que rellenarán los *parámetros formales* `base` y `exponente`.

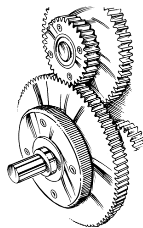
Ejemplo 10.

```
...

void main(){
  int exponente = 3;
  printf("2 elevado a 3 = %lf",
        potencia(2., exponente) );
}
```

Aunque la variable `exponente` fuese cambiada dentro de la función potencia. no habría ninguna afectación en la variable exponente de la función main. Son variables distintas con el mismo nombre, pero distinto ámbito –distinta función–. Además, lo único que "viaja" es el valor 3: no la variable exponente. Antes de poder efectuar la llamada a la función potencia, se evalúa la variable exponente y se sustituye el nombre de la variable por el valor 3. Después se escribe una copia de estos valores en los parámetros formales. *A priori, una función no puede cambiar las variables de otra.*

Mecanismo de una Llamada

El nombre de una función es equivalente a la dirección de memoria donde se encuentran definidas las instrucciones que la componen. Cuando una dirección de memoria va proseguida de unos paréntesis –vacíos o no– se produce un salto, en el flujo de ejecución, a esa dirección de memoria. La cuestiones fundamentales son: ¿cómo se sabe la dirección a la que hay que retornar? y ¿de qué manera se le puede comunicar entre funciones los parámetros y el valor de retorno?.



La evaluación del identificador de una función devuelve la dirección de memoria donde está definida.

El microprocesador tiene una serie de registros –de 16 bits inicialmente– de los cuales dos definen la pila. La pila es un espacio de memoria en el que se pueden almacernar, y del que se pueden extraer datos. Pero, sólo se puede meter un dato en la cima de la pila y, sólo se puede sacar un dato de la cima de la pila. El registro BP, *Base Pointer* indica la base de la pila y, SP, *Stack Pointer –Summit* diría yo–, la cima. La cima de la pila es la dirección de memoria de la primera posición libre.

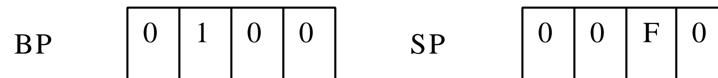


Figura 11.1: Registros de la Pila.

Podemos así imaginar una pila en la que hemos empujado los números 2,4,6 y 8, como:

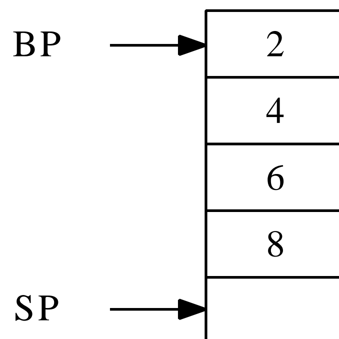
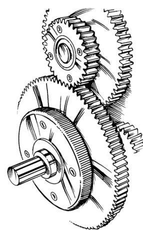


Figura 11.2: Ejemplo de pila.



La pila crece hacia hacia posiciones de memoria inferiores –hacia el 0–.

Cuando hacemos una llamada pasa lo siguiente:

1. Empujamos los parámetros actuales a la pila.
2. Empujamos la dirección de retorno –donde ha de volver el codigo–
3. Empujamos el valor de BP.
4. Movemos BP donde está SP –creamos una nueva pila para la función–.

Cuando termina la función hay que hacer lo siguiente:

1. Reestablecemos el antiguo valor de BP.
2. Saltamos al punto de retorno.
3. Sacamos los parámetros de la pila –equilibrado de la pila–.

Según se empujen los parámetros de atrás adelante o viceversa, y según si el responsable de equilibrar la pila es la función llamante o la función llamada, tenenemos diferentes *convenciones de llamada*.

11.1.4. La función main

En un programa C existe una y sólo una función de nombre main que es la que recibe el control cuando se ejecuta el programa.

Dicha función puede no devolver nada –void– en algunos sistemas operativos y, por tanto, no tiene un return . No obstante, se recomienda que la función main devuelva 0 si termina su ejecución sin incidencias y un número cualquiera en los demás casos. Escribimos así:

```
int main(){
  ...
  return 0;
}
```

Cabe criticar que 0 no es ni muy legible, ni muy indicativo de lo que ocurre. La librería stdlib define las constantes correspondientes.

Mejoramos así el código anterior.

```
#include <stdlib.h>

int main(){
  ...
  return EXIT_SUCCESS;
}
```

Cuando ejecutamos un programa escribimos su nombre en la consola de windows, o bien en linux:

`user$ ./programa`

Es posible pasar parámetros a la función main en la línea de comandos.

`user$ ./programa parametro1 parametro2`

o bien:

`user$ ./programa "parametro 1" "parametro 2"`

Estas invocaciones tienen tres parámetros. El parámetro 0 siempre es el nombre del programa que se está ejecutando –`programa`–.

Para recoger estos parámetros definimos main como:

```
#include <stdlib.h>

int main(int argc, char *argv[]){
  ...
  return EXIT_SUCCESS;
}
```

Donde argc representa el número de parámetros que tiene la invocación y argv es una lista de los mismos.

Existen parámetros adicionales de main que nos permitirían acceder a las variables de entorno del sistema operativo, pero eso: es harina de otro costal.

11.1.5. Paso por Valor y Paso por Referencia

Dos problemas dejan sin solución las funciones explicadas hasta el momento:

1. Cómo se devuelve más de un valor.

2. Cómo se puede cambiar el valor de una variable de la función llamante.

La respuesta a las dos cuestiones son los pasos por referencia. Recordemos que cuando pasamos la variable *area* en una llamada, lo que realmente ocurre es:

1. Se evalúa la variable *area*.

2. Una copia de ese valor se empuja en la pila.

3. La función llamada accede a la copia – en [BP+n]–.

Podemos localizar y cambiar una variable de la función llamante empujando en la pila su dirección, en lugar de su valor.

Para ello contamos con los operadores de dirección e indirección –* y &–.

El operador & se lee: «la dirección de». Así por ejemplo: &*area*, se lee: «la dirección de area». Si dicha variable ocupa la posición 1000 en la RAM, entonces &*area* ⇒1000.

Podemos guardar la dirección de una variable en otra. Pero, ¿de qué tipo?

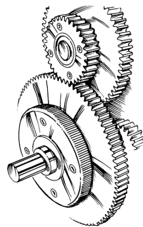
- Sea lo que sea la respuesta a la pregunta anterior, tenemos que saber que tipo de dato hay en la dirección 1000. Si hay un entero tendremos que leer dos/cuatro –dependiendo del sistema operativo– bytes a partir de 1000. Si se tratase de un char, sólo uno.

Para ello creamos el tipo de datos:

```
int *
char *
double *
...
```

que también se pueden escribir:

```
int*
char*
double*
...
```



El asterisco es totalmente indisoluble de la palabra anterior y no es un operador a parte.

Podemos definir un puntero a la variable *area* de la siguiente manera:

```
int area = 3;
int *donde;

donde = &area;
```

Decimos que *donde* apunta a *area*.

El operador * se lee: «el contenido de lo que apunta». De esta manera, *donde* ⇒1000, pero **donde* ⇒3.

Cuando pasamos la dirección de una variable en la llamada de una función, hacemos un paso por referencia.

```
#include <stdlib.h>

void incrementa(int *variable){
  *variable++;
}

int main(){
  int area = 2;

  incrementa(&area);
  printf("Area = %d", area);
  return EXIT_SUCCESS;
}
```

En este ejemplo, veremos un 3 en pantalla.

Atendiendo al otro problema planteado: que una función devuelva más de un valor. La solución es pasar dos o más variables como referencia para usarlas como cajas vacías en donde depositar los valores de retorno.

11.1.6. Recursividad

Una función es recursiva cuando dentro de su definición hay una llamada a sí misma. En tanto en cuanto cada llamada tiene su propia pila, también tiene sus propias variables locales.

Para que no sea un proceso sempiterno –con principio, pero sin fin en el tiempo–, en algún momento una llamada debe decidir no seguir llamándose a sí misma.

La recursividad consume mucha memoria, pero tiene una velocidad de ejecución extraordinaria. No todas las factorías de software permiten recursividad. Desde que aquí se anima y se valora su uso.

Para diseñar una función recursiva, hay que seguir dos pasos.

1. Expresar el problema en función de sí mismo.

2. Encontrar la condición de parada.

Valga como ejemplo encontrar el factorial de un número n.

Pongamos los factoriales de varios números:

$$
\begin{aligned}
7! &= 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\
6! &= 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\
5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1
\end{aligned}
$$

$$
\begin{aligned}
4! &= 4 \cdot 3 \cdot 2 \cdot 1 \\
3! &= 3 \cdot 2 \cdot 1 \\
2! &= 2 \cdot 1 \\
1! &= 1
\end{aligned}
$$

El factorial de 0 es 1 por convenio.
Definir el problema en función de sí mismo es darse cuenta de que:

$$
\begin{aligned}
7! &= 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\
6! &= 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\
7! &= 7 \cdot 6! \\
n! &= n \cdot (n-1)!
\end{aligned}
$$

La condición de parada es que el factorial de 0 es directamente 1. Escribimos así:

```
unsigned factorial(unsigned n){
  if (n==0)        // Condicion de parada
     return 1;
  return n * factorial(n-1);
}
```

11.1.7. Ámbito y Visibilidad

Las variables que se definen fuera de las funciones son variables globales –*extern*– existen y se conocen desde su definición hasta el final del programa.

Si una variable se define dentro de una función es una variable local –*auto*– existe y se conoce desde que se define hasta que termina la ejecución de la función. Existen dentro de la pila asignada a la función –cada llamada tiene una pila propia–.

Si la variable se declara anteponiendo la palabra reservada *static* sólo se conoce dentro de la función, pero conserva su valor entre llamadas.

```
#include <stdlib.h>

int veces(){
  static int n = 0;
  n++;
  return n;
```

```
}

int main(){
  for (int i=0; i<4; i++)
    printf("%d", veces());
  return EXIT_SUCCESS;
}
```

11.1.8. Punteros a funciones

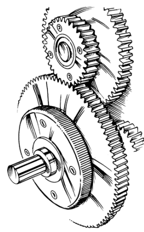
En tanto que las funciones son un conjunto de instrucciones sito a partir de una dirección de memoria asociada al nombre de la función, es posible declarar punteros a funciones. Valga como ejemplo:

```
int (*p)(int a);
```

que es un puntero a una función, la cual devuelve un entero en su llamada y debe recibir un entero como parámetro.

Esúdiese con atención la siguiente lista de ejemplos.

```
int *p;                 /* Puntero a entero */
int *p[10];             /* Lista de 10 punteros */
int (*p)[10];           /* Puntero a una lista de 10 enteros */
int *p(void);           /* Funcion que devuelve un puntero */
int p(char *a);         /* Funcion que acepta un puntero a char */
int *p(char *a);        /* Funcion que acepta un puntero a char
                           y devuelve un puntero a entero */
int (*p)(char *a);      /* Puntero a funcion que acepta un
                           puntero a char */
int (*p(char *a))[10];  /* Puntero a funcion que
                           devuelve una lista de
                           10 enteros */
int p(char (*a)[]);     /* Func que acepta un puntero
                           a una lista de char */
int p(char *a[]);       /* Func que acepta una lista
                           de punteros a char */
int (*p)(char (*a)[]);  /* Puntero a funcion que acepta
                           un puntero a una lista de char */
/* Finalmente */
int *(*p[10])(char *a)[5];
/* Lista de 10 punteros a funcion, cada una de las cuales acepta
un puntero a char y devuelve una lista de 5 enteros. */
```



Detente, reflexiona y entenderás. En la comprensión todo es evidente.