Introducción a la programación en C

1. Pasos para la resolución de un problema :

1. Definición o análisis del problema
2. Diseño del algoritmo
3. Transformación del algoritmo en un programa
4. Ejecución y validación del programa

Cabe recalcar que un algoritmo es un método para resolver un problema.

Problema → Algoritmo → Programa

2. Algoritmo

- Un algoritmo se define como una descripción general y no ambigua de los pasos necesarios para resolver cualquier problema de una clase en un tiempo finito.
- Un algoritmo no resuelve ningún problema, sólo indica cómo resolverlos.
- Para resolverlo debe existir un ente, que denominaremos procesador.
- Un algoritmo es una secuencia de sentencias que representan instrucciones para que el procesador realice acciones concretas.
- Cada procesador sólo reconoce un conjunto limitado de instrucciones.
- Para que un algoritmo sea ejecutado por un procesador, debe estar escrito utilizando instrucciones del repertorio de ese procesador.

2.1. Algoritmos entendibles por humanos:

- Los humanos tienen gran capacidad de desambiguación.
- Los programas informáticos deben escribirse en lenguajes muy restringidos (lenguajes de programación).
- Los algoritmos se escriben en pseudolenguajes similares a los lenguajes de programación, pero más flexibles.

3. Herramientas de Programación

- Diagrama de flujo
- Diagrama N-S (Nassi-Schneiderman)
- Pseudocódigo

4. Elementos de un Diagrama de Flujo

- Proceso
- Entrada/Salida
- Decisión (sí/no, múltiple)
- Conector
- Indicador de dirección

- Línea conectora
- Conector entre páginas
- Proceso predefinido
- Terminal
- Proceso manual

5. Variables

- Las variables son objetos capaces de contener “valores” que son manipulados por un procesador para resolver un problema.
- Variables en memoria según su tamaño.

Atributos de las variables

- Nombre: permite referenciarla en el programa. (relacionada con la infomación que almacenará)
- Tipo: define el tipo de valor que almacenará
- Valor: valor que almacena en cada momento de la ejecución. La capacidad de cambiar ese valor en cada momento de ejecución es lo que les da el nombre de variables.

Tipos de Variables

- Numéricos: enteros y reales, permiten operaciones aritméticas y relacionales.
- Alfanuméricos:
    - Carácter: 'a', '2', '?', etc. Operaciones: las relacionales basadas en el orden implícito
    - Ristra: "hola", "La Laguna", etc. Operaciones: Longitud, Concatenación, Localización de subristras, Extracción de subristras.
    - Lógicos: verdadero/falso. Operaciones: o, y, no.

6. Asignación y Operaciones de E/S

Asignación:

i = 30

X = i / j

A = "La Laguna"

F = Verdadero

Entrada: Leer i

Salida: Escribir "El resultado es: ", A

Declaración de Variables

- Se declara el nombre y tipo de la variable.
- Se hace antes del bloque de instrucciones ejecutables.

```
int A, B, C;
float R;
```

7. Estructuras de control

- C posee las sentencias de control típicas de los lenguajes de alto nivel.
- Las instrucciones en C pueden ser simples (expresiones o sentencias de control de flujo) o compuestas.
- main es una función especial que será la primera en ejecutarse en todo programa en C.
- El cuerpo de la función está encerrado entre llaves {}. Las llaves también determinan un bloque de código a ejecutar como unidad.
- Las variables deben declararse antes de ser utilizadas. En cualquier punto donde se abra un bloque {} se pueden definir variables locales.
- Todas las sentencias deben terminar en punto y coma ;
- Los comentarios se encierran entre /**/ y son ignorados, pudiendo aparecer en cualquier parte del código excepto en medio de un identificador

Programa sencillo en C:

```
int main() {
       int sum1=3,sum2=5,result;
       result=sum1+sum2;

       return 0;
}
```

Tipos de estructuras de control

Instrucción1;

Instrucción2;

Instrucción3;

O bien de manera lineal:

Instrucción1; Instrucción2; Instrucción3

Alternativa “If”:

La sentencia if tiene la forma:

```
if (Condición) Acción1;
[else Acción2;]
```

Los paréntesis () asociados que delimitan la condición no son opcionales. En caso de que la condición sea verdadera se ejecuta la instrucción1; en caso contrario se ejecuta, si existe, la instrucción2;

```
if (valor%2==0) par=1; /*sentencia simple de If*/

if (a>b) {
       max=a;
       cambio=1;
} else {
       max=b;
}
```

Alternativa “While”:

La sentencia while tiene la forma:

```
while (Condición) Acción1;
```

Los paréntesis no son opcionales. Si se cumple la condición se ejecuta la instrucción y se repite el proceso. Un ejemplo con el while sería:

```
expresión 1;
while (expresión 2) {
sentencia
expresión 3;
}
```

Tanto el while como el for son muy usados para recorrer arrays y cadenas de caracteres.

Repetitiva con condición final “do-while”:

```
do
     Acción1;
while (Condición);
```

Los paréntesis no son opcionales. Se ejecuta la instrucción y si se cumple la condición se repite el proceso.

```
main() {    /* visualizar los números del 0 al 9.*/
int digito=0;
do
printf("%d ",digito++);
while (digito<=9);
}
/* Visualizará: 0 1 2 3 4 5 6 7 8 9 */
```

8. Estructuras algorítmicas no fundamentales

Repetitiva for

```
for ([inicialización];[condición];[expresión])
{
     Acción1;
}
```

Cada una de las partes del for se puede omitir. En el caso de la condición, su omisión equivale a una condición siempre verdadera. Si se desea ejecutar más de una instrucción en una de las expresiones, se puede emplear el operador ,

El funcionamiento de la estructura for es el siguiente:

1) Se evalúa la expresión de inicialización.
2) Se evalúa la condición. Si el resultado es falso, termina la ejecución de la estructura for. Si es verdadero, sigue por el paso 3.
3) Se ejecuta el bloque de instrucciones dentro de la estructura.
4) Se evalúa la expresión final, que normalmente altera la variable de control, y se vuelve al paso segundo.

Proceso por "casos" switch

- La expresión entre paréntesis del switch debe ser entera. Su resultado se comparará con los distintos valores en los case. Si coincide con uno de ellos se pasará la ejecución a la instrucción siguiente al case con dicho valor y se seguirá ejecutando las instrucciones consecutivas hasta encontrar una instrucción break o alcanzar el cierra llaves del switch.
- En caso de que el resultado de la expresión no coincida con ningún valor se pasará la ejecución a la instrucción siguiente a la etiqueta default:, si la hubiera, y se continuará como en un case.En caso de que esta última no exista, continúa con la siguiente sentencia después del switch.
- Los valores en los case pueden ser una expresión constante.
- No puede haber dos case con el mismo valor.

```c
// Algoritmo Calculadora
int main()
{
    // (1 = suma, 2 = resta, 3 = multiplicación y 4 = división)

    // variables
    int i, x, y;
    float z;

    scanf("%d", &i);
    scanf("%d", &x);
    scanf("%d", &y);
```

```
    switch(i) {
        case 1:
            z = x + y;
            break;
        case 2:
            z = x - y;
            break;
        case 3:
            z = x * y;
            break;
        case 4:
            z = (float)x / y;
            break;
    }

    printf("%f", z);
}
// fin algoritmo Calculadora
```

Identificadores, tipos de datos, variables, constantes y operadores

Introducción

Programa sencillo escrito en C:

```c
int main(){
     int i,j,k; /*declaración de tres variables enteras*/
     i=13; /*se asigna un 13 (decimal) a I */
     j= i+012; //se asigna i +012(octal) a j
     k=j*j+0FF //se asigna j*j+0xFF (hexadecimal) a k*/
}
```

En los programas, la función main es la primera en ejecutarse en C, es una función especial e indica el punto de arranque del programa. A la hora de programar tenemos que tener en cuenta que todas las sentencias terminan en punto y coma (;), que las llaves "{ }" agrupan sentencias que pertenecen a un mismo bloque de código y que las variables deben ser declaradas antes de utilizarse. Se pueden declarar variables en cualquier punto donde se abra un bloque con llaves { }.

Por otro lado, los comentarios se pueden introducir añadiendo: /* */ (barra "/"asterisco "*") y delimitan los comentarios que ocupan más de una línea. Los comentarios tipo // especifican el comienzo de un comentario que acaba con el final de esa línea, es decir, solo ocupan una línea.

Es muy importante introducir comentarios en el código fuente ya que tienen como objetivo aclarar cómo está hecho el programa, facilitando la comprensión del código por parte de los programadores, ya sea para aclarar el funcionamiento de un bloque de código, documentar la lógica o recordar detalles relevantes. Cabe destacar que los comentarios son ignorados por completo por el compilador, por lo que solo ayudan a las personas a entender el código.

1. Identificadores

Antes de explicar los identificadores en C, es necesario resaltar que C es un lenguaje sensible al contexto, es decir, C nota diferencia entre mayúsculas y minúsculas, y por tanto, diferencia entre una palabra escrita total o parcialmente en mayúsculas y otra escrita completamente en minúsculas.

Los "identificadores" o "símbolos" son los nombres que se proporcionan para variables, tipos, funciones y etiquetas del programa. Los nombres de identificadores deben diferir en ortografía y mayúsculas y minúsculas de cualquier palabra clave. No se puede utilizar palabras clave (ya sea de C o de Microsoft) como identificadores; se reservan para uso especial. Para crear un identificador, hay que especificarlo en la declaración de

una variable, un tipo o una función. En este ejemplo, *result* es un identificador para una variable de tipo entero y *main* y *printf* son nombres de identificador para funciones.
Los identificadores y palabras reservadas son aquellas a las que se le asigna el nombre con el que se hace referencia a una función o al contenido de una variable o constante.
Siguen una serie de reglas:

- Están formadas por una secuencia de letras y dígitos.
- El carácter subrayado o underscore (_) es una letra más.
- No pueden contener: espacios en blanco, ni *, /,+,-,:, etc.
- No pueden comenzar con un dígito
- El primer carácter debe ser una letra o la barra baja "_"
- Se distingue entre mayúsculas y minúsculas.
- Es aconsejable usar nombres de funciones y variables que indiquen qué representa dicha función, variable o constante.

```
#include <stdio.h>
int main() {
    int result;

    if ( result != 0 )
        printf_s( "Bad file handle\n" );
}
```

A continuación se muestran las palabras reservadas del lenguaje y que por lo tanto no se pueden utilizar como identificadores: *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*.

2. Tipos de datos, modificadores de tipo y modificadores de acceso

En C, toda variable, antes de poder ser usada, debe ser declarada, especificando con ello el tipo de dato que almacenará. Toda variable en C se declara de la forma: [, nombre de variable];
En C existen cinco tipos de datos:

- char: Carácter o entero pequeño (byte)
- int: Entero
- float: Punto flotante
- double o doble: Punto flotante (mayor rango que float)
- void: Sin tipo (uso especial)

Existen, además, cuatro modificadores de tipo, los cuales se aplican sobre los tipos de datos anteriormente citados. Los modificadores de tipo permiten cambiar el tamaño, etc., de los tipos de datos anteriormente especificados. Estos modificadores, que sintácticamente anteceden a la declaración del tipo de dato, son:

- signed: char o int, Con signo (por defecto)
- unsigned: char o int, Sin signo
- long: int o double, Largo
- short: int, Corto

Así se pueden declarar variables como unsigned char a; long double b; o short int i;

Es posible, además, aplicar dos modificadores seguidos a un mismo tipo de datos, así, es posible definir una variable de tipo unsigned long int (entero largo sin signo). El rango de valores de que permite cada variable depende del sistema operativo sobre el cual se trabaje (MS-DOS, Windows95/98/NT/2000, UNIX/Linux), por lo cual conviene referirse al manual del compilador para conocerlo. De forma general, los sistemas operativos de 16 bits (MS-DOS, Windows 16 bits) poseen un rango y los de 32 bits (Windows 32 bits, UNIX/Linux) otro.

	Rango de valores posibles en (notación matemática)	
Tipo de variable declarada	**16 bits**	**32 bits**
char / signed char	[-128 , 127] (8 bits)	[-128 , 127]
unsigned char	[0 , 255]	[0 , 255]
int / signed int	[-32768 , 32767]	[-2147483647 , 2147483648]
unsigned int	[0 , 65535]	[0 , 4294967295]
short int / signed short int	[-32768 , 32767] (16 bits)	[-32768 , 32767]
unsigned short int	[0 , 65535]	[0 , 65535]
long int / signed long int	[-2147483647 , 2147483648]	[-2147483647 , 2147483648]
unsigned long int	[0 , 4294967295]	[0 , 4294967295]
float	[-3.4E+38 , -3.4E-38], 0 , [3.4E-38 , 3.4E+38] (32 bits)	[-3.4E+38 , -3.4E-38], 0 , [3.4E-38 , 3.4E+38]
double / doble	[-1.7E+308 , -1.7E-308], 0 , [1.7E-308 , 1.7E+308] (64 bits)	[-1.7E+308 , -1.7E-308], 0 , [1.7E-308 , 1.7E+308]

long double / doble	[-3.4E+4932 , -1.1E-4932], 0 , [3.4E-4932 , 1.1E+4932]	[-3.4E+4932 , -1.1E-4932], 0 , [3.4E-4932 , 1.1E+4932]
void	Dato descartado, tipo comodín	Dato descartado, tipo comodín

Ejemplos tipos de datos básicos:

```
int num; // Entero
int num1; / /Entero

short int c3; // Entero corto
short c1,c2; // Entero corto

char c; // Carácter

long int l1,l2; // Entero Largo
long L3; // Entero Largo

unsigned long int l3; // Entero Largo sin signo

float real1; // Real
double real2; // Real muy largo
```

Además de los modificadores de tipo existen modificadores de acceso. Los modificadores de acceso limitan el uso que puede realizarse de las variables declaradas. Los modificadores de acceso anteceden a la declaración del tipo de dato de la variable y son los siguientes:

- const -> Variable de valor constante
- volatile -> Variable cuyo valor es modificado externamente

La declaración de una variable como const permite asegurarse de que su valor no será modificado por el programa, excepto en el momento de su declaración, en el cual debe asignársele un valor inicial. Así, si declaramos la siguiente variable: const int x=237;

Cualquier intento posterior de modificar el valor de x, tal como x=x+5;, producirá un error en tiempo de compilación.

La declaración de una variable como volatile, indica al compilador que dicha variable puede modificarse por un proceso externo al propio programa (tal como la hora del sistema), y por ello, que no trate de optimizar dicha variable suponiéndole un valor constante, etc. Ello fuerza a que cada vez que se usa la variable, se realice una comprobación de su valor.

Los modificadores const y volatile pueden usarse de forma conjunta en ciertos casos, por lo cual no son excluyentes el uno del otro. Ello es posible si se declara una variable que actualizará el reloj del sistema, (proceso externo al programa), y que no queremos pueda modificarse en el interior del programa. Por ello, podremos declarar: volatile const unsigned long int hora;

Por otro lado, en cuanto a la Conversión entre tipos, C permite convertir valores de distinto tipo de forma automática. El lenguaje garantiza un correcto funcionamiento si el valor a convertir se encuentra dentro del rango del tipo del receptor. En otro caso, no se garantiza un buen funcionamiento.

L3=num; No hay problema, siempre que long tenga tamaño igual o mayor que int.

c=num; Sí puede generar problemas si num no está dentro del rango de char

num=real2; Sí puede generar problemas por pérdida de precisión o desbordamiento.

La conversión se puede controlar utilizando el operador de conversión cast, (tipo) expresión _a_convertir. **result=((double)num)/num2**

3. Declaración de variables y alcance

En C, las variables pueden ser declaradas en cuatro lugares del módulo del programa:

- Fuera de todas las funciones del programa, son las llamadas variables globales, accesibles desde cualquier parte del programa.
- Dentro de una función, son las llamadas variables locales, accesibles tan solo por la función en las que se declaran.
- Como parámetros a la función, accesibles de igual forma que si se declararán dentro de la función.
- Dentro de un bloque de código del programa, accesible tan solo dentro del bloque donde se declara. Esta forma de declaración puede interpretarse como una variable local del bloque donde se declara.

Para un mejor comprensión, veamos un pequeño programa de C con variables declaradas de las cuatro formas posibles:

```
#include <stdio.h>
int sum;  /* Variable global, accesible desde cualquier parte del programa */
void suma(int x) { /* Variable local declarada como parámetro, accesible solo
por la función suma
   sum=sum+x;
```

```
   return;
}
void intercambio(int *a,int *b) {
   if (*a>*b) {
      int inter; /* Variable local, accesible solo dentro del bloque donde se
declara*/
      inter=*a;
      *a=*b;
      *b=inter;
   }
   return;
}
int main(void) /*Función principal del programa*/ {
   int contador,a=9,b=0; /*Variables locales, accesibles solo por main */
   sum=0;
   intercambio(&a,&b);
   for(contador=a;contador<=b;contador++) suma(contador);
   printf("%d\n",suma);
   return(0);
}
```

4. Especificadores de almacenamiento de los tipos de datos

En este punto se explica cómo es posible modificar el alcance del almacenamiento de los datos. Esto es posible realizarlo mediante los especificadores de almacenamiento. Existen cuatro especificadores de almacenamiento. Estos especificadores de almacenamiento, cuando se usan, deben preceder a la declaración del tipo de dato de la variable.

Estos especificadores de almacenamiento son:

- auto: Variable local (por defecto)
- extern: Variable externa
- static: Variable estática
- register: Variable registro

El especificador auto se usa para declarar que una variable local existe solamente mientras estemos dentro de la subrutina o bloque de programa donde se declara, pero, dado que por defecto toda variable local es auto, no suele usarse.

El especificador extern se usa en el desarrollo de programas compuestos por varios módulos. El modificador extern se usa sobre las variables globales del módulo, de forma que si una variable global se declara como extern, el compilador no crea un almacenamiento para ella en memoria, sino que, tan solo tiene en cuenta que dicha variable ya ha sido declarada en otro modulo del programa y es del tipo de dato que se indica.

El especificador static actúa según el alcance de la variable:

- Para variables locales, el especificador static indica que dicha variable local debe almacenarse de forma permanente en memoria, tal y como si fuera una variable global, pero su alcance será el que correspondería a una variable local declarada en la subrutina o bloque. El principal efecto que provoca la declaración como static de una variable local es el hecho de que la variable conserva su valor entre llamadas a la función.
- Para variables globales, el especificador static indica que dicha variable global es local al módulo del programa donde se declara, y, por tanto, no será conocida por ningún otro módulo del programa.

El especificador register se aplica sólo a variables locales de tipo char e int. Dicho especificador indica al compilador que, caso de ser posible, mantenga esa variable en un registro de la CPU y no cree por ello una variable en la memoria. Se pueden declarar como register cuántas variables se deseen, pues el compilador ignorará dicha declaración caso de no poder ser satisfecha. El uso de variables con especificador de almacenamiento register permite colocar en registros de la CPU variables muy frecuentemente usadas, tales como contadores de bucles, etc.
Algunos ejemplos de uso de los especificadores de almacenamiento son: register unsigned int a; static float b; extern int c; static const unsigned long int d;

5. Constantes

En C, las constantes se refieren a los valores fijos que el programa no puede alterar. Algunos ejemplos de constantes de C son:

- char: ‘a’ ‘9’ 'Q'
- int: 1 -34 21000
- long int: -34 67856L 456
- short int: 10 -12 1500
- unsigned int: 45600U 345 3
- float: 12.45 4.34e-3 -2.8e9
- double o doble: -34.657 -2.2e-7 1.0e100

Existen, además, algunos tipos de constantes, distintos a los anteriores, que es necesario resaltar de forma particular. Estos tipos de constantes son las constantes hexadecimales y octales, las constantes de cadena, y las constantes de barra invertida.

Las constantes hexadecimales y octales son constantes enteras, pero definidas en base 16 (constantes hexadecimales) o en base 8 (constantes octales). Las constantes de

tipo hexadecimal comienzan por los caracteres 0x seguidos del número deseado. Las constantes de tipo octal comienzan por un cero (0). Por ejemplo, son constantes hexadecimales 0x34 (52 decimal), 0xFFFF (65535 decimal); y constantes octales 011 (9 decimal), 0173 (123 decimal).

```
0x10 => es 16 en hexadecimal
0XAB => es 171 en hexadecimal
0x12c => es300 en hexadecimal
010 =>es 8 en octal
025 =>es 21 en octal
```

Las constantes de cadena son conjuntos de caracteres que se encierran entre comillas dobles. Por ejemplo, son constantes de cadena “Esto es una constante de cadena”, “Estos son unos apuntes de C”, etc.

Las constantes de caracteres de barra invertida se usan para introducir caracteres que es imposible introducir por el teclado (tales como retorno de carro, etc.). Estas constantes son proporcionadas por C para que sea posible introducir dichos caracteres como constantes en los programas en los cuales sea necesario. Estas constantes de caracteres de barra invertida son:

\b: Retroceso
\f: Alimentación de hoja
\n: Nueva línea, salto de línea
\r: Retorno de carro
\t: Tabulador horizontal
\”: Doble comilla
\’: Simple comilla
\0: Carácter Nulo
\\: Barra invertida: Representa una “\”
\v: Tabulador vertical
\a: Alerta
\o: Constante octal
\x: Constante hexadecimal

El uso de las constantes de barra invertida es igual que el de cualquier otro carácter, así, si ch es una variable de tipo char, podemos hacer: ch=‘\t’, o ch=‘\x20’ (carácter espacio), etc., de igual forma que realizaríamos con cualquier otra constante de carácter.

Además, las constantes de barra invertida pueden usarse en el interior de constantes de cadena como un carácter más, por ello, podemos poner escribir la constante de cadena: "Esto es una línea\n".

6. Operadores

En el lenguaje de programación C, un operador es un carácter o grupo de caracteres especiales que actúan sobre una, dos o más variables y/o constantes con el fin de obtener un resultado. Los operadores permiten realizar cálculos, comparar valores, manipular bits, entre muchas otras operaciones.

Los operadores se pueden combinar entre sí para formar expresiones, las cuales siguen unas reglas de precedencia (prioridad) y asociatividad. Es posible utilizar paréntesis para modificar el orden natural de evaluación y asegurar que las operaciones se realicen en el orden deseado

Los operadores se agrupan en cinco grandes categorías:

Operadores aritméticos:

+: Suma

-: Resta

*: Multiplicación

/: División

%: Módulo (resto de la división entera o caracteres)

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

```c
int main() {
    int a = 10, b = 3;
    printf("Suma: %d\n", a + b);        // 13
    printf("Resta: %d\n", a - b);       // 7
    printf("Multiplicación: %d\n", a * b); // 30
    printf("División: %d\n", a / b);    // 3 (división entera)
    printf("Módulo: %d\n", a % b);      // 1 (resto de 10 / 3)
    return 0;
}
```

Incremento ++, decremento --:

Estos son operadores unarios que incrementan o disminuyen el valor de la variable en una unidad. Si preceden a la variable (si van antes), ésta se incrementa/decrementa antes de usar el valor de dicha variable.

```c
n=++b; // El valor original de b se incrementa y luego se asigna a n
```

Si es la variable la que precede al operador, la variable es incrementada/decrementada después de ser usada en la expresión.

```c
n=b++; // El valor original de b se asigna a n y luego se incrementa b
```

Ejemplo entero de incremento y decremento:

```c
#include <stdio.h>

int main() {
    int a, b = 0, i;
    // Primer bucle: imprime los valores de i con i++ (postincremento)
    for(i = 0; i < 10; i++) {
        printf("%i,", i);
    }
    // SALIDA: 0,1,2,3,4,5,6,7,8,9,
    printf("-----------\n");
    // Segundo bucle: imprime los valores de i con ++i (preincremento)
    for(i = 0; i < 10; ++i) {
        printf("%i,", i);
    }
    // SALIDA: 0,1,2,3,4,5,6,7,8,9,
    printf("-----------\n");
    // Tercer bucle: analiza el uso de b++ (postincremento) a toma el valor
actual de b, luego b se incrementa
    for(i = 0; i < 10; ++i) {
        a = b++;  // primero asigna b a a, luego incrementa b
        printf("a=%i--", a);
        printf("b=%i", b);
        printf("\n");
    }
    /*
    SALIDA:
    a=0--b=1
    a=1--b=2
    a=2--b=3
    ...
    */
    printf("-----------\n");
    b = 0; // Se reinicia b a 0
    // Cuarto bucle: analiza el uso de ++b (preincremento) b se incrementa antes
de ser asignado a a
    for(i = 0; i < 10; i++) {
        a = ++b;  // primero incrementa b, luego lo asigna a a
        printf("a=%i--", a);
        printf("b=%i", b);
        printf("\n");
    }

    /*
    SALIDA:
    a=1--b=1
    a=2--b=2
    ...
    */
```

```
    return 0;
}
```

Operadores relacionales:

Se utilizan para comprobar la veracidad o falsedad de determinadas propuestas de relación (en realidad se trata de respuestas a preguntas). Las expresiones que los contienen se denominan expresiones relacionales. Aceptan diversos tipos de argumentos, y el resultado, que es la respuesta a la pregunta, es siempre del tipo cierto/falso, es decir, producen un resultado booleano. Se utilizan para comparar dos valores. El resultado siempre es verdadero (1) o falso (0). Estos son:

== Igual a

!= Distinto de

> Mayor que

< Menor que

>= Mayor o igual que

<= Menor o igual que

```c
#include <stdio.h>
int main() {
    int x = 5, y = 10;
    printf("¿x == y? %d\n", x == y); // 0 (falso)
    printf("¿x != y? %d\n", x != y); // 1 (verdadero)
    printf("¿x > y? %d\n", x > y);   // 0
    printf("¿x < y? %d\n", x < y);   // 1
    printf("¿x >= y? %d\n", x >= y); // 0
    printf("¿x <= y? %d\n", x <= y); // 1
    return 0;
}
```

Operadores lógicos:

Permiten combinar los resultados de los operadores relacionales. Sus operandos son también valores lógicos o asimilables a ellos (los valores numéricos son asimilados a cierto o falso según su valor sea cero o distinto de cero).

&& Y lógico (AND)

|| O lógico (OR)

! Negación (NOT)

```c
#include <stdio.h>
int main() {
    int a = 1, b = 0;
    printf("AND lógico: %d\n", a && b); // 0 (1 y 0 = falso)
    printf("OR lógico: %d\n", a || b);  // 1 (al menos uno es verdadero)
    printf("NOT lógico: %d\n", !a);     // 0 (negación de verdadero)
    return 0;
```

```
}
```

Operadores de asignación:

Atribuyen a una variable el resultado de una expresión o el valor de otra variable.

Sirven para asignar valores a variables. El más básico es el =, pero también existen combinaciones con operadores aritméticos:

= Asignación

+=, -=, *=, /=, %= Asignaciones compuestas

= : a=3; a=b; (a=b=c àa=c:b=c)

+= : a+=5 à a=a+5)

-= : a-=3*b à a=a-(3*b)/

***=** : a*=2 à a=a*2

/= : a/=35+b à a=a/(35+b)

%= : a%5àa=a%5

```c
#include <stdio.h>
int main() {
    int a = 5;
    a += 3; // Equivale a: a = a + 3;
    printf("a += 3 → %d\n", a); // 8
    a *= 2; // a = a * 2
    printf("a *= 2 → %d\n", a); // 16
    return 0;
}
```

Tabla de Precedencia y Asociatividad:

Operadores	*Asociatividad*
() [] -> .	izquierda a derecha
! ~ ++ -- + - * & (tipo) sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha

^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= /= %= &= ^= = <<= >>=	derecha a izquierda (todos juntos)
,	izquierda a derecha

7. Estructura de un programa

```
/*
    Nombre del programa y del proyecto.
    Descripción:
    Autor y fecha:
*/
/* Directivas al preprocesador */
#include <libreria> // Bloque de librerías
#define <nombre> <valor> // Bloque de macros
// Comienzo del cuerpo principal:
int main(){
    Bloque de declaración e inicialización de variables <tipo_de_dato>
<lista_de_variables> ;
    Bloque de instrucciones: asignación, lectura, escritura, condicionales,
iterativas
}
```

8. Estilo de programación

Para desarrollar programas en lenguaje C de manera clara, eficiente y fácil de mantener, es importante seguir un conjunto de convenciones de estilo. Estas buenas prácticas mejoran tanto la comprensión del código como su calidad general.

Comentar el código es fundamental y obligatorio. Siempre se debe explicar qué hace cada parte del programa mediante comentarios adecuados. Es recomendable declarar una sola variable por línea y añadir un comentario que indique su propósito o función en el programa. Esto facilita el mantenimiento y la comprensión del código por parte de otros programadores.

Keep It Simple, Stupid: La simplicidad debe ser una prioridad. Aplicar el principio “Keep It Simple, Stupid” (KISS) implica escribir código directo y sin complicaciones innecesarias. Asimismo, se recomienda seguir la “Ley del mínimo asombro”, que sugiere que el

comportamiento del código debe ser lo más predecible posible. El objetivo es que otro programador pueda leer el código y entenderlo sin confusión.

En cuanto al uso de operadores, es aconsejable evitar escribir expresiones complejas que utilicen ++ o --. Lo ideal es emplearlos en líneas independientes para mantener la claridad. También se debe evitar colocar asignaciones (=) dentro de condiciones (if, while, etc.), ya que puede confundirse con el operador de igualdad (==) y provocar errores difíciles de detectar.

Desde el punto de vista de la legibilidad, se deben colocar espacios después de cada coma y punto y coma. Además, se recomienda dejar un espacio en blanco a ambos lados de los operadores binarios como +, -, =, entre otros, para facilitar la lectura del código.

La interfaz con el usuario debe mantenerse lo más simple y clara posible. Es importante que todos los mensajes, especialmente los de error, estén bien identificados y ofrezcan información precisa y útil.

En relación con los identificadores, se deben usar nombres descriptivos que reflejen claramente el propósito de variables y funciones. Evite el uso de nombres genéricos o abreviados sin sentido. Los identificadores de macros y constantes deben escribirse en mayúsculas, mientras que los nombres de variables y funciones deben ir en minúsculas.

Respecto al uso de estructuras de control como switch, es una buena práctica incluir siempre una cláusula default, incluso si no realiza ninguna acción. Todas las opciones del switch deben finalizar con una instrucción break para evitar una ejecución no deseada del siguiente caso. En cuanto a la sintaxis, se recomienda colocar la llave de apertura { al final de la línea de la estructura de control correspondiente, y no en una línea aparte. Asimismo, es esencial aplicar una indentación uniforme y clara en todo el programa, ya sea utilizando espacios o tabulaciones.

Seguir estas pautas no solo mejora la calidad del código, sino que también facilita su revisión, depuración y mantenimiento a largo plazo. Un código bien escrito es una herramienta poderosa y un reflejo del cuidado y profesionalismo del programador.

```c
#include <stdio.h>

int main() {
    int edad = 0; // Edad del usuario
    char nombre[50]; // Nombre del usuario
```

```
    printf("Ingrese su nombre: ");
    scanf("%s", nombre);

    printf("Ingrese su edad: ");
    scanf("%d", &edad);

    if (edad >= 18) {
        printf("%s es mayor de edad.\n", nombre);
    } else {
        printf("%s es menor de edad.\n", nombre);
    }

    return 0;
}
```

La directiva #include en C indica al preprocesador que incluya el contenido de un archivo específico en el código fuente del programa antes de la compilación. En este ejemplo, #include <stdio.h> indica al preprocesador que incluya el archivo stdio.h, que contiene la declaración de la función printf().

En resumen: #include es una directiva del preprocesador que permite incluir el contenido de un archivo (generalmente un archivo de cabecera) en el código fuente, facilitando la organización, reutilización y modularidad del código en C.

Biblioteca de funciones

<stdio.h>

<math.h>

En el lenguaje C, no existen instrucciones incorporadas para entrada y salida (E/S) como parte del núcleo del lenguaje. En su lugar, estas operaciones se realizan a través de funciones definidas en bibliotecas estándar, en ese caso, **<stdio.h>.** Dos de las bibliotecas más utilizadas para estas tareas son **<stdio.h> y <math.h>**.

1. Biblioteca <stdio.h>

La biblioteca estándar de entrada/salida (<stdio.h>) es fundamental en cualquier programa C. Proporciona las funciones necesarias para mostrar datos por pantalla y para leer datos introducidos por el usuario. Las dos funciones más comunes de esta biblioteca son **printf()** y **scanf()**, pero también hay otras funciones que se utilizarán en la asignatura o en las prácticas:

- getchar lee un carácter
- putchar escribe un carácter
- gets lee una ristra
- puts escribe una ristra

Teniendo en cuenta esto, se tendrán como más importantes las siguientes:

La función **printf()** se utiliza para mostrar información por pantalla. Su primer argumento es una cadena de caracteres (también llamada ristra), que puede incluir texto literal y especificadores de formato. Estos especificadores, precedidos por el símbolo %, indican cómo deben mostrarse los valores de las variables que se pasan como parámetros. Por ejemplo:

```
printf("a = %i, b = %o, c = %x", a, b, c);
```

En este caso, a se imprime en formato decimal, b en octal y c en hexadecimal. Algunos de los formatos más comunes que se pueden usar con printf() son:

%d o %i: entero decimal

%o: entero en octal

%x: entero en hexadecimal

%u: entero sin signo

%f: número en coma flotante

%e: notación científica

%s: cadena de caracteres

%c: un solo carácter

%ld, %lu, %lx: variantes para enteros largos

También es posible utilizar secuencias de escape en la cadena de formato para imprimir caracteres especiales. Estas secuencias comienzan con \ e incluyen:

- \n: salto de línea
- \t: tabulador horizontal
- \v: tabulador vertical
- \b: retroceso (borra el carácter anterior)
- \r: retorno de carro (vuelve al inicio de la línea)
- \a: alerta sonora
- \': comilla simple
- \": comilla doble
- \\: barra invertida
- \?: signo de interrogación
- \000: carácter cuyo código ASCII está en octal
- \xhh: carácter en hexadecimal (ASCII)

Por otro lado, la función **scanf()** se emplea para leer datos de entrada introducidos por el usuario. Su uso es muy similar al de printf(), pero en este caso los parámetros deben ser punteros a las variables donde se almacenarán los datos leídos. Por ejemplo:

```
scanf("%i", &x);
```

Aquí, scanf() interpreta la entrada como un número entero y lo almacena en la dirección de memoria de la variable x.

printf-scanf ejemplos:

```
/* Uso de la sentencia scanf() y printf()*/

#include <stdio.h>

main() /* Solicita dos datos */
{
	char nombre[10];
	int edad;
	printf("Introduce tu nombre: ");
	scanf("%s",nombre);
	printf("Introduce tu edad: ");
	scanf("%d",&edad);
}
```

2. Biblioteca <math.h>

La biblioteca <math.h> proporciona una serie de funciones matemáticas que operan sobre valores reales (tipo double). Estas funciones cubren una amplia gama de cálculos matemáticos, desde funciones trigonométricas hasta logaritmos y raíces. Algunas de las funciones más importantes son:

- double sin(double x), cos(double x), tan(double x): funciones trigonométricas básicas
- double asin(double x), acos(double x), atan(double x): funciones trigonométricas inversas
- double exp(double x): exponencial de x (e^x)
- double log(double x): logaritmo natural
- double log10(double x): logaritmo en base 10
- double pow(double x,double y): x elevado a la potencia y
- double sqrt(double x): raíz cuadrada de x
- double fabs(double x): valor absoluto de x

Para utilizar cualquiera de estas funciones, es necesario incluir la cabecera <math.h> y en muchos compiladores también hay que enlazar la biblioteca matemática usando la opción -lm durante la compilación (por ejemplo: gcc programa.c -lm).

Punteros y vectores

Punteros

En el lenguaje de programación C, un puntero es un tipo de dato especial que no almacena directamente valores como los enteros o los caracteres, sino que guarda **direcciones de memoria**. En otras palabras, un puntero es un objeto que apunta a otro objeto. Esto significa que su valor no es un dato tradicional, sino la ubicación en memoria donde se encuentra almacenada otra variable.

Para declarar un puntero, se utiliza la siguiente sintaxis:

```
tipo_de_dato * variable_puntero;.
```

Por ejemplo, int *p; declara un puntero que apunta a una variable entera. Es importante saber que un puntero solo puede almacenar direcciones de memoria correspondientes al tipo de datos para el que ha sido declarado. Es decir, un puntero a entero solo puede apuntar a variables enteras.

Variables y punteros

Cuando se declara una variable en C, el sistema **reserva** un espacio en memoria para almacenar su valor. Para poder acceder a esa variable, el compilador necesita conocer dos cosas: el **número de bytes** que ocupa (según su tipo de dato) y la **dirección de memoria** del primer byte reservado. Esta dirección se puede obtener utilizando el nombre de la variable.

Operadores relacionados con punteros

En C existen dos operadores clave para trabajar con punteros:

- Operador de dirección (&): permite obtener la dirección de memoria de una variable. Por ejemplo, si tenemos long dato;, al escribir &dato obtendremos la dirección donde se encuentra almacenado el valor de dato.

```
long dato;
printf("%lu", &dato);
int *p;
p = &k;
```

- Operador de indirección (*): se utiliza para acceder al contenido de la dirección de memoria apuntada por un puntero. Por ejemplo, si int *p; y p = &x;, entonces *p nos da el valor almacenado en x.

```
int *p;
*p = 3;
```

Ejemplo manejo punteros

Este programa muestra cómo modificar valores a través de punteros y cómo intercambiar el contenido de dos variables.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int x = 4, y = 7, temp;
    int *px = NULL, *py = NULL; //Declaración de punteros a enteros e inicio
    printf(" \nx es %d y es %d ",x,y);
    px = &x; // px apunta a la dirección de la variable x
    py = &y; // px apunta a la dirección de la variable y
    printf("\npx contiene la direccion %p, py contiene la direccion %p ", px,
py) ;
    printf( "\nLa direccion de la variable x es %p, la direccion de la variable
y es %p", &x , &y) ;
    *px = *px + 10; //Modificamos el contenido de la dirección de memoria
apuntada por px
    *py = *py + 10; //Modificamos el contenido de la dirección de memoria
apuntada por py
    printf(" \nx es %d y es %d ",x,y) ;
    // Intercambiando valores
    temp = *px ;
    *px = *py ;
    *py = temp ;
    printf("\npx contiene %d, py contiene %d ",*px,*py) ;
    printf(" \nx es %d y es %d ",x,y);
    system("PAUSE") ;
    return 0;
}
```

Precauciones al usar punteros

Es fundamental inicializar los punteros antes de usarlos. Acceder al contenido de una dirección de memoria mediante un puntero no inicializado puede provocar errores graves durante la ejecución del programa, como violaciones de segmento. Para evitarlo, se recomienda inicializar los punteros con NULL o con 0 (que equivale a una dirección nula). Así, si se intenta acceder a través de un puntero sin haberle asignado una dirección válida, el error será más fácil de detectar.

```
Tipo_de_dato * variable_puntero = 0;
o bien
Tipo_de_dato * variable_puntero = NULL;
```

De esta manera, si se intenta acceder a esta dirección de memoria antes de asignar otro valor se producirá un error fácil de identificar en tiempo de ejecución (violación de segmento).

Tipos de datos estructurados: Arrays unidimensionales estáticos

Un array (también llamado vector) es una colección de elementos del mismo tipo almacenados de forma contigua en memoria. En C, el tamaño de un array debe conocerse en tiempo de compilación y no puede cambiar durante la ejecución del programa.

Los arrays tienen dos propiedades importantes:

- El índice del primer elemento siempre es 0, por lo que un array de N elementos va desde el índice 0 hasta N-1. V[0]
- El nombre de un array actúa como un puntero constante que apunta al primer elemento del array. Por esto, no se pueden asignar arrays entre sí directamente.
  Como consecuencia de la segunda propiedad, no es posible asignar directamente un vector a otro.

Arrays unidimensionales

- Declaración: tipoDato nombre_array[N_ELEMENTOS];

```
int alturas [40] ;
float a[20], b[100];
```

- El número de elementos debe ser una expresión constante entera.
- A cada elemento del array se accede a través de un índice. El índice inferior en C es siempre 0 y el superior es N_ELEMENTOS – 1

```
alturas[39]
```

- Sus elementos pueden ser de cualquier tipo básico

```
char nombre[10] ;
```

- Es común el uso de macros para acotar el máximo número de elementos a almacenar.

```
#define N_ELEMENTOS 40
char nombre[N_ELEMENTOS] ;
```

Cuestiones sobre los arrays de caracteres: cadenas

En C, las cadenas son arrays especiales de caracteres, es decir son un tipo particular de vector. Se pueden inicializar con una cadena literal, es decir, una secuencia de caracteres encerrada entre comillas dobles. Estas cadenas terminan automáticamente con el carácter nulo ('\0'), aunque no lo veamos. Por ejemplo, la cadena "Primero" ocupa en realidad 8 caracteres (7 visibles más el '\0'), por lo que necesitaríamos un array de tamaño

8 o superior para almacenarla correctamente. también soporta caracteres especiales mediante secuencias de escape como: \n (nueva línea), \t (tabulación), \b (retroceso), \\ (barra invertida), entre otros.

Tipos de datos estructurados: Arrays multidimensionales

Las matrices, se representan como arrays de arrays. En C se pueden construir matrices de cualquier dimensión, lo que nos permite representar estructuras como matrices.

Para acceder a una posición se escribe el nombre y a continuación el índice entre corchetes para cada dimensión.

Se declara de la siguiente manera:

```
tipo_Dato nombre[DIM1][DIM2]...[DIMN];
int matriz[20][10]; /*filas, columnas*/
```

Funciones y procedimientos

Programación modular

La programación modular es una técnica que consiste en dividir un programa en subprogramas o módulos independientes que pueden incluso almacenarse en archivos distintos. Esto permite que tareas complejas se descompongan en subproblemas más pequeños, que pueden ser tratados y resueltos de manera aislada. Cada uno de estos subproblemas puede implementarse como una función o procedimiento, lo que favorece la reutilización del código y facilita su comprensión.

Para poder llevar a cabo la programación modular, será necesario modificar el código, tal y como lo hemos visto hasta ahora, para conseguir que se comporte de manera independiente. Para ello, deberán presentar dos características:

1. Debe poder ser invocado desde otro algoritmo, lo que se consigue definiendo correctamente su cabecera.
2. Debe poder recibir información del exterior (datos de entrada) y devolver resultados (datos de salida). Esto se consigue mediante la utilización de parámetros formales.

Parámetros formales

Los parámetros formales son variables que se utilizan para recibir datos que el algoritmo principal pasa al módulo, o para devolver resultados al mismo. Dependiendo de su función, estos parámetros se pueden clasificar en:

- Parámetros de entrada, que reciben valores desde el algoritmo llamador.
- Parámetros de entrada/salida, que reciben datos y también almacenan los resultados para que el algoritmo llamador pueda acceder a ellos.

Además, dentro de una función o procedimiento también se pueden declarar variables locales, que no forman parte de la lista de parámetros y cuya utilidad se limita al funcionamiento interno de la función. Algunas de estas variables locales pueden servir como parámetros de salida, es decir, variables donde se almacenan resultados que serán devueltos al final de la ejecución del módulo.

Variables locales

Las variables de la función/procedimiento que no figuran en la lista de parámetros, reciben el nombre de variables locales y se utilizan exclusivamente para operaciones internas, sin relación con el algoritmo llamador. El valor de algunas de ellas pueden, almacenar

resultados y ser devueltas para que los recoja el llamador. Se conocen como: Parámetros de salida.

Ventajas de la programación modular

La modularidad tiene múltiples ventajas. Una de las más importantes es que cada módulo puede ser depurado (revisado y corregido) de forma independiente, lo que facilita la localización de errores. Además, permite que un equipo de desarrolladores trabaje simultáneamente, repartiendo los distintos módulos del programa. Otra ventaja clave es la reutilización del código, ya que un módulo bien diseñado puede ser empleado en varios programas sin necesidad de modificarlo.

Definición de funciones

En el lenguaje C no existen los procedimientos como tal; todas las subrutinas se definen como funciones. Cuando una función no devuelve ningún valor, se utiliza el tipo void, lo que equivale al comportamiento de un procedimiento en otros lenguajes.
Las funciones se definen indicando primero que tipo de valor devuelven, después el nombre de la función y a continuación los parámetros que acepta encerrados entre paréntesis y separados por comas. Esto conformaría la cabecera de la función.
El cuerpo de la función, encerrado entre llaves { }, donde se encuentran las instrucciones a ejecutar.

```
tipo_devuelto nombre_función (tipo1 param1, tipo2 param2,...)
{
...
}
```

```
int sumar(int a, int b) {
    return a + b;
}
```

Paso de parámetros

En C, los parámetros se pasan siempre por valor, lo que significa que se copia el contenido de la variable al parámetro de la función. Esto implica que cualquier modificación que se haga dentro de la función no afecta a la variable original.

Sin embargo, C permite simular el paso por referencia utilizando punteros. En este caso, el parámetro de la función se define como un puntero al tipo correspondiente, y al llamar a la función se pasa la dirección de memoria de la variable. De esta forma, cualquier modificación dentro de la función afectará directamente a la variable original.

Ejemplo de paso por valor (no intercambia los valores de verdad):

```c
#include <stdio.h>
#include <stdlib.h>
void permutar(double x, double y)
{
double temp ;
temp = x ;
x = y ;
y = temp ;
}

int main()
{
double a = 1.0, b = 2.0 ;
printf("\na = %lf , b = %lf", a, b) ;
permutar(a, b) ;
printf("\na = %lf , b = %lf", a, b) ;
system("PAUSE");
return 0 ;
}
```

Ejemplo de paso por referencia (sí intercambia los valores):

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

void permutar(double *x, double *y)
{
double temp ;
temp = *x ;
*x = *y ;
*y = temp ;
}

int main()
{
double a = 1.0, b = 2.0 ;
printf("\na = %lf , b = %lf", a, b) ;
permutar(&a, &b) ;
printf("\na = %lf , b = %lf", a, b) ;
system("PAUSE");
return 0 ;
}
```

Paso de arrays como parámetros

El identificador de un array representa la dirección de memoria del primero de sus elementos. Por lo que en realidad se está pasando por referencia (sin necesidad de

anteponer el símbolo &) y no por valor. Si se desea que los elementos del array no puedan modificarse, se utiliza el modificador const.

```
void por_valor(const float temperaturas[])
```

En el parámetro formal NO se indica el tamaño del array:

```
tipo Funcion(int elm[]);
```

Para que la función conozca el tamaño del array que se le pasa, se usa un segundo parámetro.

```
tipo funcion(int elm[], int n_elem );
```

Uso del modificador const

El modificador const se utiliza para indicar que una variable no debe ser modificada. Si se intenta cambiar su valor, el compilador generará un error. Esto es especialmente útil en funciones para garantizar que ciertos parámetros se usan solo como entrada y no se alteran.

Parámetros de entrada/salida

C, no dispone de parámetros de entrada y salida. La forma de simular estos parámetros es definiéndolos como punteros. Donde el parámetro se define como un puntero al tipo de la variable entrada/salida y en la llamada a la función se pasa la dirección de la variable que se desea modificar.

Definición de funciones

En C, una función debe estar definida antes de ser utilizada. Si no es posible definirla antes, se debe declarar un prototipo. Un prototipo consiste en la cabecera de la función seguida por un punto y coma ;, sin incluir el cuerpo. Así informamos al compilar que existe una función con ese nombre y que será definida más adelante.
La forma de realizar el prototipado es similar a la definición pero utilizando solo la cabecera, omitiendo el cuerpo y termiando en ";"

```
int sumar(int, int);  // Prototipo
```

La sentencia return

La sentencia return se usa para devolver un valor desde una función al algoritmo llamador. Esta sentencia puede aparecer en cualquier punto de la función y cuantas veces sea necesaria y su ejecución hace que la función termine inmediatamente. En funciones de

tipo void, return se puede usar simplemente para salir de la función sin devolver nada. En el caso de los procedimientos se puede utilizar
return para abandonarlo en cualquier punto.

Resumen del paso de parámetros en C

Parámetro especificado	Item pasado por	¿Cambia item dentro de la función?	Modifica los valores de los argumentos asociados a la llamada?
int ítem	Valor	Sí	No
const int ítem	Valor	No	No
int *ítem	Referencia	Sí	Sí

Ristras, struct

Ristras de caracteres

En el lenguaje de programación C no existe un tipo de dato específico para representar cadenas de caracteres, como sucede en otros lenguajes. En su lugar, las ristras (o cadenas de caracteres) se representan como vectores de tipo char, es decir, arrays cuyos elementos son caracteres. Cada uno de los caracteres de la cadena se almacena en una posición del array, y al final de la secuencia de caracteres debe incluirse un carácter especial que indica el final de la ristra: el carácter nulo, representado como '\0'.

La diferencia fundamental entre un simple vector de caracteres y una ristra es precisamente la inclusión de este carácter nulo al final, que permite a funciones del lenguaje como printf, strlen, strcpy, etc., saber cuándo termina la cadena.

Uso de literales de ristras

Los literales de ristras se definen como secuencias de caracteres encerradas entre comillas dobles, por ejemplo: "Hola mundo". Estos literales, cuando no forman parte de una inicialización directa de un array, son tratados como punteros (char *) a una zona de memoria donde el compilador almacena la cadena. Es importante tener en cuenta que el programador no debe modificar directamente el contenido de estas zonas de memoria, ya que están gestionadas por el compilador.

Por ejemplo, si declaramos un puntero de tipo char *s; y luego asignamos s = "Texto literal";, lo que estamos haciendo es apuntar el puntero s a una zona de memoria donde se almacena esa cadena. En cambio, si declaramos un array como char c[] = "Texto literal";, entonces cada carácter de la cadena (incluido el carácter nulo) se copia dentro del array.

Esto implica que el tamaño del array debe ser igual al número de caracteres de la cadena más uno, para incluir el '\0'.

Estructuras (struct) en C

Una estructura o struct en C es un tipo de dato compuesto que agrupa varias variables en una sola unidad lógica. Cada una de las variables internas se conoce como campo, y cada campo puede ser de un tipo distinto. Las estructuras permiten organizar datos complejos y relacionados bajo un mismo nombre.

La definición de una estructura se realiza con la palabra clave struct, seguida del nombre de la estructura y de los campos que contiene. Por ejemplo:

```
struct InfoPersonal {
    long DNI;
    char Nombre[30];
};
```

Esta estructura representa una persona con dos campos: un número de DNI (long) y un nombre que puede tener hasta 29 caracteres (más el carácter nulo final). Una vez definida, podemos crear variables de este tipo:

```
struct InfoPersonal persona;
struct InfoPersonal lista[20];
```

Para facilitar la declaración, se puede usar un alias de tipo:

```
typedef struct {
    long DNI;
    char Nombre[30];
} InfoPersonal;
```

Acceso a los campos de una estructura

Para acceder a los campos de una estructura se utiliza el operador punto (.). Por ejemplo:

```
persona.DNI = 12345678;
persona.Nombre[0] = 'A';
```

Si se tiene un array de estructuras, se puede acceder a los campos de un elemento específico usando su índice:

```
lista[3].DNI = 87654321;
```

Uso de punteros con estructuras y el operador ->

Cuando se utiliza un puntero a una estructura, hay dos formas de acceder a sus campos:

- Usando el operador * (de indirección) junto con el operador punto ((*puntero).campo)

```
(*p).DNI = 45678901;
```

- O de manera más habitual, utilizando el operador flecha (->), que es una notación más compacta y legible (puntero->campo).

```
p->DNI = 45678901;
```

Ambas formas son equivalentes, pero la segunda es preferida por su simplicidad. Este uso es fundamental cuando se trabaja con estructuras dinámicas o estructuras pasadas como argumentos a funciones por referencia.