Maestría en Tecnología de Cómputo

Curso propedéutico de Programación

Tema I: Introducción al lenguaje C.

Conferencia No. 1

Título: Introducción al lenguaje C. Conceptos elementales. Operadores y expresiones.

Objetivo: Introducir conceptos iniciales de programación en C para la implementación de algoritmos de estructura secuencial.

Sumario:

1. Introducción

Una computadora es una máquina electrónica digital compuesta por tres componentes principales: procesador, memoria y dispositivos de entrada/salida. El procesador es también llamado unidad de procesamiento digital o CPU por sus siglas en inglés y es el elemento que interpreta las instrucciones y procesa los datos de los programas. Los dispositivos de entrada/salida garantizan el flujo de información entre la máquina y agentes externos. Entre ellos se encuentran los teclados, monitores, disco duros, ratones, manejadores de CD y DVD, etc. La memoria interna, también conocida como memoria de acceso aleatorio o RAM (*Random Access memory*) por sus siglas en ingles, se encarga de almacenar el código y los datos de los programas que el procesador está ejecutando en ese momento.

Las computadoras electrónicas constituyen potentes herramientas que ayudan al hombre a resolver problemas de diversa naturaleza. En aras de lograr este objetivo, las computadoras deben ser programadas para que puedan desarrollar la solución a dichos problemas. Los informáticos se encargan de analizar, diseñar e implementar estos programas o aplicaciones también conocidos como software.

En la asignatura se estudiará cómo programar las computadoras para resolver problemas sencillos y de mediana complejidad, para lo cual se abordarán las estructuras básicas de

programación, algunos tipos de datos y algoritmos básicos, así como, otros conceptos muy importantes para un principiante en la materia.

El tema que comienza en esta conferencia, se ocupa del estudio de elementos básicos de programación que permitirán en lo adelante escribir programas para ser ejecutados en una computadora. Se estudiarán sentencias básicas de los lenguajes de programación y la sintaxis de cada una se analizará en el marco del lenguaje C que ha sido el lenguaje escogido para la introducción de los conceptos de programación.

2. Escribir, compilar y ejecutar un programa

A los programas escritos en lenguajes de programación se les llama también código fuente. Una vez que el código fuente ha sido creado con ayuda de algún editor de texto, se invoca a un programa llamado compilador cuya función es traducir un lenguaje de alto nivel a otro de más bajo nivel que la máquina sea capaz de interpretar.

Como muestra la figura 2.1, una vez que el programador escribe el código fuente, este debe ser compilado y probado. Si se requiere hacer algunas modificaciones, el código fuente deberá ser editado otra vez, de manera que se entra en un ciclo que termina cuando el programador esté satisfecho.

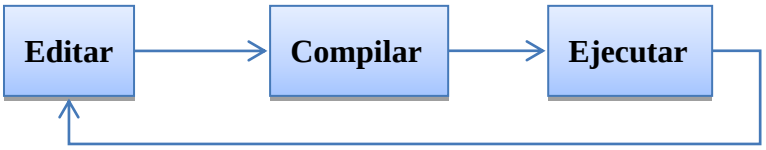


Figura 2.1. Ciclo de desarrollo de un programa

3. El lenguaje C

El lenguaje C fue desarrollado en 1972 en los laboratorios Bell de AT&T, por Dennis Ritchie, como una modificación del lenguaje B. el C es un lenguaje de alto nivel, que tiene como ventajas que el código producido por su compilador está muy optimizado. Sin embargo, el C es independiente de la plataforma en que se ejecuta solamente en su código fuente, cada plataforma debe proporcionar un compilador adecuado.

4. Un primer programa

Una tarea muy sencilla para comenzar a programar en C pudiera ser la de imprimir en pantalla la frase: Hola Mundo. A continuación se muestra el programa:

```
/********************* Hola Mundo ********************************/
// mi primer progrma en C

#include <stdio.h>
```

```
int main(int argc, char **argv)
{
        printf("Hello world\n");
}
```

La salida de este programa es:

```
Hola mundo!!!
```

El programa comienza con dos líneas de comentarios. Todo texto que comience con un par de diagonales (slash) `//` es un comentario. Un comentario es una nota al programador y no tiene que ver con el funcionamiento del programa, no es traducido por el compilador de C a ninguna instrucción ejecutable.

Un comentario que comienza con `//` termina en el final de la línea. De modo que

```
//Si esto pretende ser un comentario será un error ya que el compilador de C
pretende interpretar esta segunda línea como una instrucción de C
```

Si un comentario ocupa más de una línea puede escribirse

```
// Este comentario ocupa
// dos líneas
```

Hay una forma más cómoda para expresar comentarios de más de una línea, se empieza el comentario con los símbolos `/*` y se termina con `*/`

```
/* Este comentario ocupa más de una línea y
es correcto */
```

Posteriormente, en la tercera línea, se incluye un fichero de cabecera studio.h. Este fichero contiene la información de las funciones de entrada y salida. La palabra reservada `#include` se utiliza para identificar la inclusión de un cierto fichero. Todas las palabras reservadas que comienzan con # se llaman directrices del compilador, y son procesadas antes de iniciarse la compilación del programa.

Una palabra reservada o palabra clave es una palabra con un uso específico dentro del lenguaje y que no puede utilizarse para otro fin.

El método `main` está seguido por paréntesis, dentro de ellos se indican los parámetros utilizados por el método, pero en este caso no se utiliza ningún parámetro. Una aplicación que contenga una clase con un método `main()` significa que la ejecución de la aplicación comenzará por ejecutar a dicho método. Todo programa en C tiene una función principal, llamada main.

A su vez la implementación del método (lo que se conoce también como cuerpo del método) va encerrada entre dos llaves `{` y `}`.

Este tipo de programa se conoce como aplicación de consola porque interactúa con el usuario mediante una ventana llamada consola. La información que fluye a través de esta es en forma de texto.

La siguiente línea del método `Main` es `printf("Hola mundo!!! \n");`

La función `printf` pertenece a la biblioteca de C, y su objetivo es escribir en pantalla la expresión que aparece entre comillas “”.`\n` es una secuencia de escape que indica al cursor que se desplace al inicio de la siguiente línea. las líneas de instrucciones finalizan con un ;.

5. Identificadores

Al igual que en cualquier lenguaje de programación, en C un identificador no es más que, como su propio nombre indica, un nombre con el que identificaremos algún elemento de nuestro código, ya sea una clase, una variable, un método, etc.

Típicamente el nombre de un identificador será una secuencia de cualquier número de caracteres alfanuméricos –solamente los del alfabeto inglés- tales que el primero de ellos no sea un número. En C se distingue mayúsculas de minúsculas, por lo tanto la variable **A** es diferente de la variable **a**, así que se debe ser cuidadosos en ese sentido. Además, se permite el carácter de subrayado _.

Identificadores permitidos:

```
iNroEntero
_Valor
multiplo1
Arriba
C3P0
```

Identificadores NO permitidos:

```
while  //es un palabra reservada
1_variable    //comienza con un número
porsentaje%   //contiene caracteres especiales
@      //es solamente un carácter especial
C#      // # indica inicio de directiva de preprocesador
a!b    // ! indica operación lógica “not”
```

Una buena práctica de programación es colocar como identificador palabras descriptivas, como puede ser sueldo del mes, concatenando estas y poniendo la primera letra en mayúscula salvo la primera Ej: sueldoDelMes.

6. Secuencias de escape

Una secuencia de escape está formada por el carácter \ seguido de una letr o una combinación de dígitos. Son utilizados para acciones especiales.

Secuencia	Uso
\n	Ir al principio de la siguiente línea
\t	Tabulador horizontal

\v	Tabulador vertical (para uso de la impresora)
\b	Retroceso (*backspace*)
\r	Retorno de carro sin avance de línea
\f	Alimentación de página (para uso de la impresora)
\a	Alerta, pitido
\’	Comilla simple
\”	Comilla doble
\\	Barra invertida (*backslash*)
\ddd	Carácter ASCII. Representación octal (d es un dígito 0-7)
\xdd	Carácter ASCII. Representación hexadecimal (d es un dígito 0-7 o una letra A-Z)

Tabla 6.1. Secuencias de escape

7. Palabras Reservadas

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

Tabla 7.1. Palabras reservadas

Las palabras reservadas son aquellas que tienen un significado gramatical especial para el lenguaje y no pueden ser utilizadas como un identificador.

Por ejemplo, en C como en muchos lenguajes de programación, una variable no puede ser llamada *double*, porque la palabra *double* es usada para indicar que un identificador se

refiere a un tipo de dato real. Al tratarse de una palabra reservada su uso queda restringido. La Tabla 7.1 muestra todas las palabras reservadas del lenguaje C.

8. Tipos de datos

Los tipos de datos en C pueden agruparse en dos categorías generales, los tipos primitivos y los tipos derivados. Los tipos primitivos se presentan integrados al lenguaje. Hay siete tipos de datos primitivos (**`char`**, **`short`**, **`int`**, **`long`**, **`enum`**, **`float`** y **`double`**). Los cinco primeros son tipos enteros, y los dos restantes son tipos reales. Los tipos derivados son construidos a partir de los tipos primitivos. Algunos de ellos son estructuras, uniones, matrices (arreglos), punteros y funciones. Serán estudiados en próximas clases.

8.1. Tipos de datos primitivos

Los tipos primitivos que representan enteros son char, short, int, long y enum; y los tipos para representar números reales o de punto flotante son **`float`** y **`double`**. Cada tipo entero puede ser calificado como signed (con signo, ya sea este positivo o negativo) o unsigned (sin signo, se asume que el valor es positivo). Si los calificadores signed y unsigned se utilizan sin un tipo, se asume el tipo int. La tabla 8.1 muestra los tipos por valor predefinidos de C así como una breve descripción:

Tipo	Descripción	Ejemplo

`char`	Tipo entero de 8 bits con signo, rango de valores de -128 a 127. Incluye los valores ASCII, ANSI o UNICODE	`char a = 97;`
		`char a = 'a';`
		`char a = 0x61;`

`short`	Tipo entero de 16 bits con signo, rango de valores de -32768 a 32767	`short a = 97;`
`int`	Tipo entero de 32 bits con signo, rango de valores de –2147483648 a 2147483647	`int a = 97;` `int a = -97;`
`long`	Tipo entero de 64 bits con signo, rango de valores de –9223372036854775808 a 9223372036854775807	`long v1 = 12;` `long v2 = 34L;`
`unsigned short`	Tipo entero de 16 bits sin signo, rango de valores de 0 a 65535	`unsigned short v1 = 97;`
`unsigned int`	Tipo entero de 32 bits sin signo, rango de valores de 0 a 4294967295	`unsigned int v1 = 97;` `unsigned int v2 = 34U;`

`unsigned long`	Tipo entero de 64 bits sin signo, rango de valores de 0 a 18446744073709551615	`unsigned long v1 = 12;`
		`unsigned long v2 = 34U;`
		`unsigned long v3 = 56L;`
		`unsigned long v4 = 78UL;`

enum	Representa una lista de valores	`enum dia {lu,ma,mi};`
		`enum dia ayer;`
		`ayer = ma;`
		`ayer = 1;`

float	Tipo de punto flotante de simple precisión (32 bits) , rango aproximado de valores de 1.5×10^{-45} a 3.4×10^{38} con una precisión de 7 dígitos.	`float v = 1.23F;`
double	Tipo de punto flotante de doble precisión (64 bits), rango aproximado de valores de 5.0×10^{-324} a 1.7×10^{308} con una precisión de 15-16 dígitos	`double v1 = 1.23;` `double v2 = 4.56D;`

Tabla 8.1. Tipos de datos Primitivos

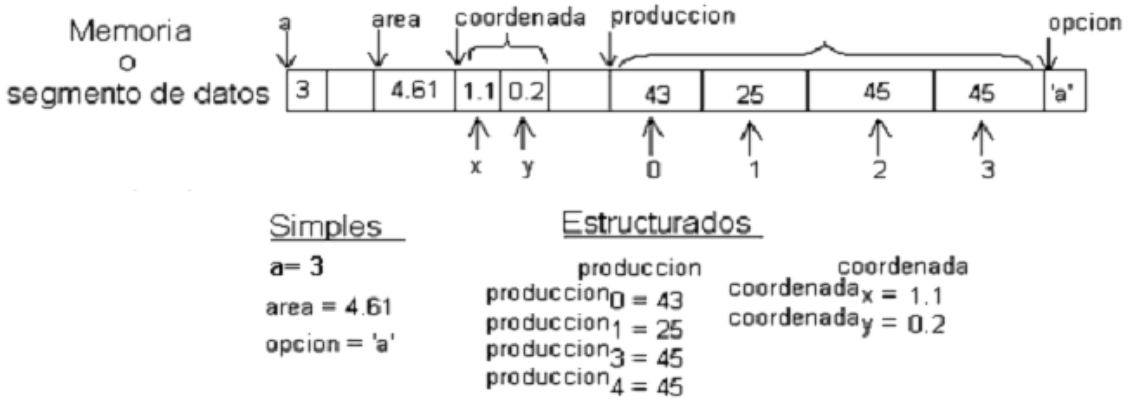
8.2. Sinónimos de un tipo

Es posible declarar nuevos nombres de tipos de datos, usando la palabra reservada `typedef`. Es decir, se pueden declarar sinónimos de los tipos de datos, ya sean primitivos o derivados. Por ejemplo, se puede definir el tipo ulong como sinónimo del tipo de datos unsigned long.

```
typedef unsigned long ulong;
ulong v1 = 17;
```

9. Variables

En los lenguajes de programación se le llama variable a todos los identificadores a través de los cuales se pueden almacenar datos en la memoria durante la ejecución de los programas y acceder a ellos. Toda variable tiene un identificador, un tipo y un valor. Las variables pueden referenciar datos simples o datos estructurados lo cual depende de la forma en que haya sido declarada la variable. El siguiente esquema ilustra de manera gráfica el almacenamiento de algunas variables en memoria:



Nótese que a cada variable se asocia una zona de memoria de un tamaño suficiente, según el tipo de dato, para almacenar sus valores.

Para declarar las variables se usa el identificador de tipo de la variable seguido del identificador de variable. La sintaxis (regla de escritura) es:

<identificador de tipo> <identificador de variable>

donde:

identificador de tipo: identificador de tipo estándar o algún otro tipo previamente definido por el programador, que permite al compilador conocer la memoria que debe reservar para el uso de dicha variable y además hacer chequeos de compatibilidad durante la ejecución del código.

identificador de variable: es el nombre, escogido por el programador para hacer referencia, durante el cuerpo del código, al valor almacenado en la zona de memoria correspondiente a la variable.

La declaración de variable, en C, se realiza donde se usa por primera vez la variable o en cualquier parte del cuerpo previo a que sea usada por primera vez. Una misma variable debe ser declarada una sola vez. El alcance de una variable empieza en su declaración y se extiende hasta la llave derecha de cierre (**}**) del bloque donde se declaró. Por lo tanto, enunciados anteriores a la declaración de la variable no pueden hacer referencia a ella, aun si están dentro del mismo bloque. De la misma manera no pueden ser referenciadas por enunciados posteriores al cierre del bloque.

Cuando se hace la declaración de una variable se está indicando al compilador que en tiempo de compilación reserve en la memoria interna el espacio necesario para guardar el valor de dicha variable. El espacio reservado sólo será usado para este fin durante la ejecución de la aplicación y a través del identificador de la variable se podrá acceder a él ya sea para guardar o recuperar la información. A este tipo de reservación se le llama **reservación estática**.

A diferencia de las constantes, las variables pueden tomar diferentes valores durante la ejecución del código. Cada vez que una variable cambia su valor se actualiza la zona de la memoria donde ella se almacena, por lo que el valor almacenado allí anteriormente se pierde.

Ejemplos

```
float diametroInterior;

int cantidadTrabajadores;

char consonante;

char esVisible;
```

En la variable *diametroInterior*, se pueden almacenar valores reales, mientras que en *cantidadTrabajadores* se pueden almacenar números enteros. La variable no numérica *Consonante* permitirá manipular caracteres (uno en cada instante de tiempo). *esVisible* podrá guardar cualquier carácter, considerando que n valor diferente de cero se corresponde con el valor `true` y un valor igual a cero se corresponde con el vlor `false`.

Si en algún momento de la ejecución del código se trata de almacenar en una de estas variables un valor no permitido el compilador generará un mensaje de error y se detendrá la ejecución. A este chequeo realizado por el compilador se le denomina chequeo de compatibilidad de tipos.

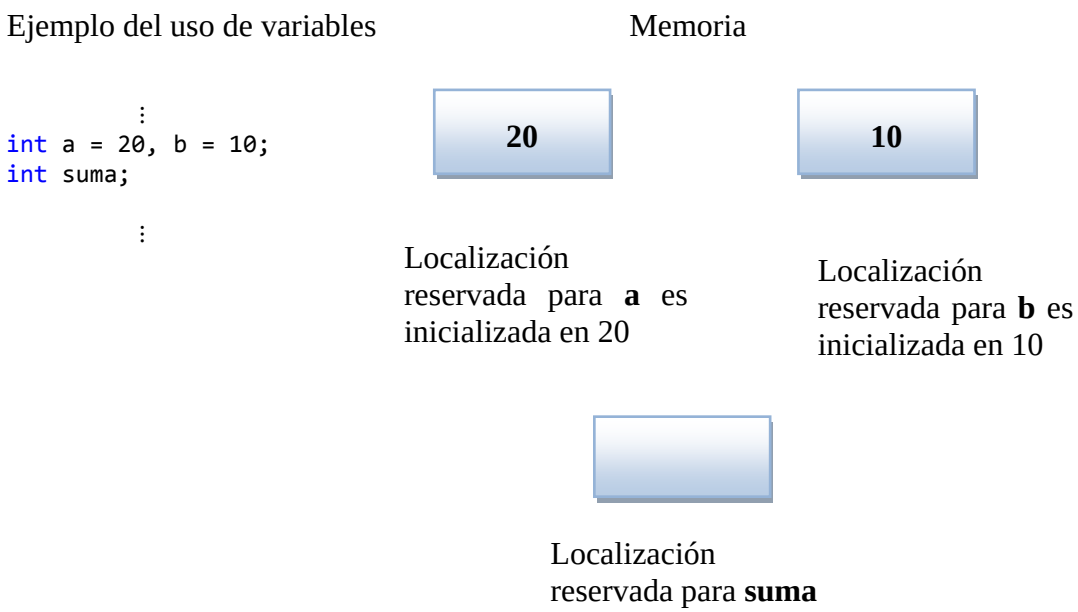
Las variables en C pueden ser inicializadas con un valor en la declaración como se muestra a continuación:

```
int cantidadTrabajadores = 100;
```

En una declaración pueden declararse más de una variable, en ese caso se usa la coma "," para separar las variables entre sí. Todas las variables serán del tipo especificado al inicio de la declaración. Ejemplo:

```
float r, s = 1.5;
```

En este ejemplo se reserva espacio para una variable flotante *r* y para otra variable flotante *s*. Esta última se inicializa con el valor `1.5`.



Nótese del ejemplo que la declaración es la que indica que se haga la reservación de espacio en la memoria. Por tanto:

- No se puede usar una variable que no se haya declarado previamente.
- Se reservarán tantos bytes como requiera el tipo especificado en la declaración.

Es necesario explicar que aunque el programador no asigne ningún valor a una variable, una vez que ésta se ha declarado, el espacio de memoria quedará reservado y no se puede asumir que está vacío. Por el contrario, allí siempre habrá algún valor. Una forma de comprobar esto es declarando una variable e inmediatamente mostrar su contenido. Hay algunos compiladores que inicializan las variables en valores por defecto, por ejemplo las numéricas en 0, las lógicas en false, etc., pero muchos no lo hacen, por lo que una buena práctica es asumir como responsabilidad del programador inicializar las variables siempre

que sea necesario. En el ejemplo anterior se ha representado que el programador no ha asignado ningún valor a la variable *suma*, no que su localización esté vacía.

10. Constantes

Una **constante** es una variable cuyo valor puede determinar el compilador durante la compilación y puede aplicar optimizaciones derivadas de ello. Para que esto sea posible se ha de cumplir que el valor de una constante no pueda cambiar durante la ejecución, por lo que el compilador informará con un error de todo intento de modificar el valor inicial de una constante. Las constantes se definen como variables normales pero precediendo el nombre de su tipo del modificador `const` y dándoles siempre un valor inicial al declararlas. O sea, con esta sintaxis:

#define <nombreConstante> <valor>

Así, ejemplos de definición de constantes es el siguiente:

```
#define a 123
#define b a + 125
```

Dadas estas definiciones de constantes, lo que hará el compilador será sustituir en el código generado todas las referencias a las constantes `a` y `b` por los valores 123 y 248 respectivamente, por lo que el código generado será más eficiente ya que no incluirá el acceso y cálculo de los valores de **a** y **b**. Nótese que puede hacer esto porque en el código se indica explícitamente cual es el valor que siempre tendrá **a** y, al ser este un valor fijo, puede deducir cuál será el valor que siempre tendrá **b**.

11. Literales

Cuando queremos expresar de manera directa un valor, estamos expresando lo que se conoce como un valor literal, o simplemente un literal. Los literales pueden ser utilizados en cualquier contexto en que pueda utilizarse una expresión que calcule un valor.

11.1. Literales numéricos

En el caso de los literales de tipo numérico estos pueden, de manera opcional, estarán acompañados de un sufijo que permite especificar el tipo de dato de dicho literal. Los literales numéricos pueden dividirse en dos categorías: los literales enteros y los literales reales. Los primeros incluyen a los tipos numéricos enteros y los últimos a los literales de tipo `float` o `double`.

Los literales enteros se pueden escribir de tres formas posibles, usando la notación decimal (en base 10), en notación octal (en base 8) o en notación hexadecimal (en base 16). Los literales enteros decimales se representan de la siguiente forma:

<dígitos-decimales sufijo$_{opc}$>

donde los dígitos decimales son los dígitos comprendidos desde el 0 hasta el 9. Por ejemplo 12,54 y 257 son literales enteros decimales. Si un literal es decimal y no tiene sufijo, su tipo será el primero de los tipos `int`, `long int` o `unsigned int` en el que su valor pueda ser representado.

En cambio los literales enteros hexadecimales se representan de la siguiente manera:

<0x dígitos-hexadecimales sufijo$_{opc}$>

Los dígitos hexadecimales representados con las letras del alfabeto pueden escribirse tanto en mayúsculas como en minúsculas. Por ejemplo `0x65ed`, `0xFFFF`, `0x4238`son literales enteros en notación hexadecimal. El sistema de numeración hexadecimal es en base a 16 (que es 2^4 es decir una potencia de 2) y tiene utilidad en aplicaciones que necesiten de forma fácil trabajar con números según su representación interna en la computadora.

Si el literal es octal o hexadecimal y no tiene sufijo, su tipo será el primero de los tipos `int`, `unsigned int`, `long int` o `unsigned long int` en el que su valor pueda ser representado.

Por otra parte los literales reales, que son los de tipo `float o double`, se expresan de las siguientes formas:

<dígitos-decimales. dígitos-decimales parte-exponente$_{opc}$sufijo$_{opc}$>

o

< . dígitos-decimales parte-exponente$_{opc}$sufijo$_{opc}$>

donde <**parte exponente**> es opcional y está compuesta por:

<esigno$_{opc}$dígitos-decimales> o <Esigno$_{opc}$dígitos-decimales>

Donde **< signo >** puede ser `+` o `-`.

Los sufijos de tipo real son `f` o `F` para el tipo **`float`** `y d` o `D` para el tipo **`double`**. Algunos ejemplos son:

```
3.14159f    // literal de tipo float
1.23e-15d  // literal de tipo double
.542D      // literal de tipo double
```

Como se mencionaba, para los literales de tipo entero si no se especifica un sufijo, el tipo que se tomará es el tipo **`int`**. Si no es posible representarlo en un **`int`**, se tratará de representar con un **`unsigned int`**, y así sucesivamente se prueba con los tipos **`long`** y

unsignged long. De esta forma, el tipo de un literal sin sufijo será el primero de los cuatro tipos integrales mencionados anteriormente, donde pueda ser representado ese literal. En cuanto a los literales de punto flotante sin sufijo se toma que son de tipo **double**. En la siguiente tabla se muestran los sufijos posibles:

Sufijos	Tipos
u, U	Es el primero de los dos tipos siguientes donde pueda representarse el literal: `unsigned int`, `unsigned long`.
l, L	Es el primero de los dos tipos siguientes donde pueda representarse el literal: `long`, `unsigned long`.
UL, Ul, ul, uL, LU, Lu, lU, lu	Es el tipo de datos `unsigned long`.
f, F	Es para literales de tipo `float`.
d, D	`d, D` Es para literales de tipo `double`.

Tabla 11.1. Tabla de sufijos

11.2. Literales de carácter

Prácticamente cualquier carácter se puede representar encerrándolo entre comillas simples. Por ejemplo, 'a' (letra a), ' ' (carácter de espacio), '?' (símbolo de interrogación), etc. Las secuencias de escape son consideradas como un único carácter. Ejemplo: '\x07' representa un pitido.

11.3. Literales de cadena

Una cadena no es más que una secuencia de caracteres encerrados entre comillas dobles. Por ejemplo: `"Hola mundo!!!"`. El texto contenido dentro estos literales puede estar formado por cualquier número de literales de carácter concatenados.

La siguiente tabla recoge algunos ejemplos de cómo se interpretan:

Literal de cadena	Interpretado como...
`"Hola Mundo"`	`Hola Mundo`
`""`	`//cadena vacía`

Tabla 11.2. Ejemplos de literales de cadena planos

12. Operadores

Un operador en C es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.

A continuación se describen cuáles son los operadores incluidos en el lenguaje clasificados según el tipo de operaciones que permiten realizar.

12.1. Operadores aritméticos

Los operadores aritméticos incluidos en C son los típicos de suma (**+**), resta (**–**), producto (*****), división (*/*) y módulo (**%**).

Cuando en una operación aritmética los operandos son de diferentes tipos, el resultado de la operación se convierte al tipo del operando con precisión más alta. En una operación de asignación, el resultado es convertido implícita o explícitamente al tipo de la variable que almacenará dicho resultado.

12.2. Operadores lógicos

Se incluyen operadores que permiten realizar las operaciones lógicas típicas: "and" (**&&**), "or" (**||**), "not" (**!**) y "xor" (**^**). Los operadores **&&** y **||** realizan evaluación perezosa. La evaluación perezosa consiste en que si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente. Es decir, si el primer operando de una operación **&&** es falso se devuelve `0` directamente, sin evaluar el segundo; y si el primer operando de una **||** es cierto se devuelve `1` directamente, sin evaluar el otro.

12.3. Operadores relacionales

Estos son los tradicionales operadores de igualdad (**==**), desigualdad (**!=**), "mayor que" (**>**), "menor que" (**<**), "mayor o igual que" (**>=**) y "menor o igual que" (<=). Los operandos deben ser de tipos primitivos.

12.4. Operadores de manipulación de bits

Son los operadores que permiten realizar a nivel de bits operaciones "and" (**&**), "or" (**|**), "not" (**~**), "xor" (**^**), desplazamiento a izquierda (**<<**) y desplazamiento a derecha (**>>**) El operador **<<** desplaza a izquierda rellenando con ceros, mientras que el tipo de relleno realizado por **>>** depende del tipo de dato sobre el que se aplica: si es un dato con signo mantiene el signo, y en caso contrario rellena con ceros.

12.5. Operadores de asignación

Para realizar asignaciones se usa en C el operador **=**, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión ***a = b*** asigna a la variable ***a*** el valor de la variable ***b*** y devuelve dicho valor, mientras que la expresión ***c = a = b*** asigna a ***c*** y ***a*** el valor de ***b*** (el operador **=** es asociativo por la derecha).

El tipo de la variable de la izquierda del operador tiene que ser compatible al del resultado de la evaluación de la expresión de la derecha, de lo contrario el compilador generará un error de incompatibilidad de tipos.

También se han incluido operadores de asignación compuestos que permiten ahorrar tecleo a la hora de realizar asignaciones tan comunes como:

```
temperatura = temperatura + 15;   // Sin usar asignación compuesta
temperatura += 15;          // Usando asignación compuesta
```

Las dos líneas anteriores son equivalentes, pues el operador compuesto **+=**lo que hace es asignar a su primer operando el valor que tenía más el valor de su segundo operando. Como se ve, permite compactar bastante el código.

Aparte del operador de asignación compuesto **+=**, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: +=, -=, *=, /=, %=, &=, |=, ^=, <<= y >>=. Nótese que no hay versiones compuestas para los operadores binarios **`&&`** y **`||`**.

Otros dos operadores de asignación incluidos son los de incremento (**++**) y decremento (**--**). Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican. Así, estas líneas de código son equivalentes:

```
temperatura = temperatura + 1;  // Sin usar asignación compuesta ni incremento
temperatura += 1;           // Usando asignación compuesta
temperatura++;                    // Usando incremento
```

Si el operador **++** se coloca tras el nombre de la variable (como en el ejemplo) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras incrementarla; y lo mismo ocurre con el operador **--**. Por ejemplo:

```
c = b++; // Se asigna a c el valor de b y luego se incrementa b
c = ++b; // Se incrementa el valor de b y luego se asigna a c
```

La ventaja de usar los operadores **++** y **--** es que en muchas máquinas son más eficientes que el resto de formas de realizar sumas o restas de una unidad, pues el compilador los traduce en una única instrucción en código máquina.

12.6. Operador condicional

También conocido como operador ternario. Se representa por ?. Su sintaxis es:

operando1 ? operando2 : operando 3

operando1 debe ser una expresión booleana (que toma valores verdadero o falso). Si el resultado de la evaluación de operando1 es verdadero, el resultado del operador condicional es operando2. En caso contrario, el resultado del operador condicional será operando3.

12.7. Otros operadores

Operador sizeof: devuelve el tamaño en bytes de operando. El operando puede ser el identificador o tipo de una variable. Si el operando es un tipo de datos, la sintaxis correcta es sizeof(operando). De lo contrario, puede aplicarse como sizeof operando.

Operador coma: se utiliza para encadenar varias operaciones, que se ejecutan de izquierda a derecha. Ejemplo: int aux = 1, aux = 2;

Operador dirección-de: se utiliza para conocer la dirección de memoria de su operando. Se representa por &. Ejemplo: &a devuelve la dirección de memoria de la variable a.

Operador de indirección: Se representa por *. Se utiliza para acceder a un valor indirectamente a través de una dirección (un puntero). Devuelve el valor direccionado por el operando, es decir, el valor apuntado por el puntero.

Un puntero es una variable capaz de contener una dirección de memoria que indica dónde se localiza un dato de tipo especificado. Un puntero se define como <tipo*identificador>.

Ejemplo:

```
#include <studio.h>
main()
  {
       int *px;
       int x = 7, y = 0;
       px = &x;
       y = *px;
       printf("Dirección de memoria = %d, dato = %d\n", &x, x);
       printf("Dirección de memoria = %d, dato = %d\n", px, *px);
  }
```

La salida de este programa es:

Dirección de memoria = 1245048, dato = 7

Dirección de memoria = 1245048, dato = 7

Nótese que una vez asignada la dirección de x a px, *px y x hacen referencia al mismo dato.

13. Expresiones

Es un conjunto de operandos combinados entre sí, a través de operadores. A toda expresión se asocia un valor que depende del valor de cada operando en el momento de su evaluación y de las operaciones indicadas entre ellos por los operadores.

Las expresiones según el resultado de su evaluación pueden ser:

- Numéricas, al evaluarlas se obtiene un valor numérico producto de la combinación de operandos numéricos a través de operadores aritméticos.
- Lógicas, al evaluarlas se obtiene un valor lógico, producto de la combinación de operandos del mismo tipo a través de operadores de relación, o de otras expresiones combinadas a través de operadores lógicos.

Ejemplos de expresiones

```
x + y * z
Alfa * 22 – h
PI * Radio * Radio
(h + s) / (n * q)
```

13.1. Precedencia y asociatividad de los operadores

Los operadores tienen reglas de precedencia y asociatividad que determinan como la expresión es evaluada. Al evaluar una expresión se realizan las operaciones de izquierda a derecha, y según la precedencia de los operadores que aparecen en ella. Para violar la precedencia de los operadores, se pueden utilizar paréntesis ( ) dentro de las expresiones, con las mismas reglas conocidas del álgebra. No se pueden usar para esto los corchetes, pues están reservados para otro uso en el lenguaje. La tabla contiene la precedencia y asociatividad de los operadores. Se muestran en la misma fila los operadores de igual prioridad, y en orden de prioridad (cada fila tiene mayor prioridad que la siguiente)

`() [] . ->`	**Izquierda a derecha.**
`- ~ ! ++ -- sizeof`	**Derecha a izquierda.**
`* / %`	**Izquierda a derecha.**
`+ -`	**Izquierda a derecha.**
`<< >>`	**Izquierda a derecha.**
`< > <= >= instanceof`	**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.**

Tabla 13.1. Precedencia y asociatividad de los operadores

14. Estructuras básicas de la programación

Al inicio de la década del 60 comenzaron a aparecer trabajos sobre la programación estructurada. De manera general, todos de una forma u otra coincidían en afirmar que para escribir códigos claros y correctos sólo se necesitaban tres estructuras de programación:

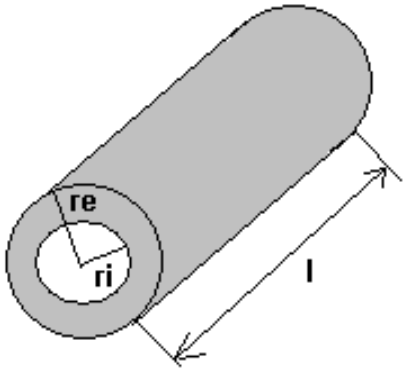
a) secuencial
b) alternativa
c) repetitiva

La *estructura secuencial*, es simplemente la idea de que las sentencias de un código son ejecutadas en el orden en que aparecen en él. A través de estructuras secuenciales se implementan fragmentos lineales de los algoritmos.

Ejemplo de estructura secuencial

Problema: Veamos como programar una secuencia que calcule el volumen de un cilindro hueco conocidos su radio interior, su radio exterior y longitud.

Primero analicemos el modelo matemático:



Área del círculo:

$a = \prod . r^2$

Volumen del cilindro

V = Área de la base. Longitud = $\prod .r^2 .l$

Área de la base:

AB = Área Círc. Ext. - Área Círc. Int

Volumen del cilindro hueco:

VCH = AB .l

```
/*el siguiente fragmento corresponde a una secuencia que calcula el volumen de un
cilindro hueco conocidos su radio interior, su radio exterior y su longitud*/

#include <stdio.h>
#define PI 3.1416f

int main()
{
	int radioExterior = 0;
	int radioInterior = 0;
	int longitud = 0;
	printf("Entre el radio exterior: \n");
	scanf("%d", &radioExterior);
	printf ("Entre el radio interior: \n");
	scanf("%d", &radioInterior);
	printf ("Entre la longitud: \n");
	scanf("%d", &longitud);
	float areaBaseExterior = PI * radioExterior * radioExterior;
	float areaBaseInterior = PI * radioInterior * radioInterior;
	float areaBase = areaBaseExterior - areaBaseInterior;
	float volumen = areaBase / longitud;
	printf("El volumen del cilindro hueco es: %g\n", volumen);
}
```

En el ejemplo, se utiliza el método `scanf` para obtener los valores de las medidas del cilindro escritos sobre el teclado por los usuarios. Este método hace detener el programa en espera que el usuario teclee un dato y presione la tecla Enter para introducir el dato a la computadora. Más adelante se abordará en detalle el manejo de instrucciones de entrada/salida (I/O).

Seudocódigos.

El seudocódigo es un lenguaje cómodo y amigable, similar al inglés coloquial; que auxilia a los programadores a desarrollar los algoritmos, que luego serán traducidos a un lenguaje verdadero de programación de computadoras.

En la siguiente tabla se muestran un resumen de las operaciones vistas en esta conferencia y sus equivalentes en seudocódigo.

Seudocódigo	C	Significado
Operadores Lógicos.		
>	>	Mayor que
<	<	Menor que
=	==	Igual a

<=	<=	Menor o igual que
>=	>=	Mayor o igual que
<>	**!=**	Distinto
OR	||	Suma lógica (O)
AND	&&	Producto lógico (Y)
NOT	!	Negación (NO)
Operadores Aritméticos.		
+	+	Suma
-	-	Resta
*	*	Multiplicación
/	/	División
MOD	%	Resto de la división entera
Asignaciones		
←	=	Asignación
Entrada y salida de información		
IMPRIMIR “cadena”	`printf`	Salida por consola.
LEER cadena	`scanf`	Entrada desde consola

Hasta el momento hemos podido resolver problemas sencillos programando un simple algoritmo que describe paso a paso lo que debemos hacer, pero hay situaciones más complejas en que es más razonable diseñar los algoritmos, basados en el principio del diseño descendente (top down) o modular, según el cual los problemas complejos se resuelven dividiendo el problema en problemas más simples.

A partir de ahora vamos a empezar a intentar descomponer los problemas en trozos más pequeños, que sean más fáciles de resolver lo que supone varias ventajas:

- Cada "trozo de algoritmo" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.

- Podremos repartir el trabajo, para que cada persona se encargue de realizar un "trozo de algoritmo", y finalmente se integrará el trabajo individual de cada persona.

La modularidad se logra en C a través de la implementación de bloques de códigos conocidos por métodos.

15. Métodos.

En algunos lenguajes de programación como en C suele llamársele funciones, en otros orientados a objetos como el C++ se distinguen ambos términos. Métodos se le dice a aquellas funciones que forman parte del contexto de una clase.

Un método es un bloque de código que contiene una serie de instrucciones. Los programas hacen que las instrucciones se ejecuten mediante una llamada al método y la especificación de los argumentos de método necesarios. En C, cada instrucción se ejecuta en el contexto de un método. El método main es el punto de entrada de cada aplicación cuando se inicia el programa.

15.1. Declaración de un método.

La declaración de un método es el momento en que se define el identificador del método, su tipo de retorno, los parámetros a través de los cuales intercambia información con el resto de la aplicación y se hace explícito todo lo que es capaz de hacer, o sea, se escribe su código completo.

La sintaxis general es:

<TipoRetorno> <IdentificadorMétodo> (**<TipoParámetro> <IdentificadorParámetro>**,…)

{

<secuencia de declaraciones y sentencias>

}

Donde:

<TipoRetorno> : Es el tipo del valor que devuelve el método una vez que es invocada. Si el método no devuelve nada, esto se indica con la palabra reservada void.

<IdentificadorMétodo>: Es el nombre del método donde se puede poner cualquier identificador válido.

<TipoParámetro> y <IdentificadorParámetro> : Es el tipo y el identificador del parámetro o argumento que se le pasa. En caso de que el método tenga más de un parámetro estos van separados por coma.

Acerca de la declaración:

- La lista de parámetros formales que se escribe opcionalmente a continuación del identificador, constituye la interfaz de comunicación del método con el exterior.
- No está permitido declarar métodos dentro de otros métodos.

- Las instrucciones a ejecutar constituyen el cuerpo del método y se escriben entre llaves ({}).

Por ejemplo el siguiente método calcula el volumen de un cilindro hueco recibiendo como datos el radio exterior, el radio interior y la longitud. O sea, veremos cómo hacer un método con el fragmento de código que se hizo en la conferencia anterior.

```
float VolumenCilindroHueco(float radioExterior, float radioInterior, float longitud)
{
   float volumen = 0;
   float areaBaseExterior = PI * radioExterior * radioExterior;
   float areaBaseInterior = PI * radioInterior * radioInterior;
   float areaBase = areaBaseExterior - areaBaseInterior;
   volumen = areaBase * longitud;

   return volumen;
 }
```

Como se puede observar:

- El método recibe los datos *radioExterior*, *radioInterior* y *longitud* a través de sus parámetros, esto nos permite calcular el volumen del cilindro hueco para distintos valores de radios y longitud, pues éstos entrarían a través de los parámetros.
- Los parámetros se declaran en el encabezamiento, por lo que no es necesario volver a declarar dichos identificadores en el cuerpo del método.
- Todos los identificadores declarados dentro del método son locales. O sea, no pueden ser accedidos desde el exterior del método.
- En el cuerpo del método hay una sentencia **return**. Esta sentencia es la encargada de retornar el valor que devuelve el método, por tanto la expresión que sigue a la palabra reservada **return** tiene que ser del mismo tipo que el tipo del método, o de algún tipo compatible con él.
- Una vez que se ejecuta la sentencia **return** se abandona el método.

15.2. Llamada a un método.

Los métodos se invocan desde algún otro lugar de la aplicación ya sea desde otro método o directamente desde el método principal *Main*. En este momento se transfiere el control al punto donde está declarada la función y se ejecuta desde el principio hasta el fin todo su código. Una vez que se haya completado la ejecución, el flujo del control regresará al punto donde ocurrió la invocación.

Los métodos se llaman usando la notación siguiente:

<**nombreMétodo**>**(**<**valoresParámetros**>**)**

El formato en que se pasen los valores a cada parámetro en **<valoresParámetros>** a aquellos métodos que tomen parámetros, depende del tipo de parámetro que sea. Esto se explica en el siguiente apartado.

16. Parámetros de método y Argumentos.

La definición de método especifica los nombres y tipos de cualquier parámetro que se requiera. Cuando el código de llamada llama al método, proporciona valores concretos denominados argumentos a cada parámetro. Los argumentos deben ser compatibles con el tipo de parámetro pero el nombre del argumento (si existe) que se usa en el código de llamada no tiene que ser igual que el nombre de parámetro definido en el método. Por ejemplo:

```
void Invocador()
{
    float radioExterior1 = 4;
    float radioInterior1 = 2;
    float longitud1 = 10;
    // Sus argumentos son variables
    float volumen1 = VolumenCilindroHueco(radioExterior1, radioInterior1, longitud1);
       printf("El volumen del cilindro hueco es: %g\n", volumen1);
    float radioExterior2 = 10;
    float radioInterior2 = 8;
    float longitud2 = 20;
    float volumen2 = VolumenCilindroHueco(radioExterior2, radioInterior2, longitud2);
       printf("El volumen del cilindro hueco es: %g\n", volumen2);
       float radioExterior3 = 10;
    float radioInterior3 = 2;
    float longitud3 = 10;
    // Dos de sus argumentos son expresiones
    float volumen3 = VolumenCilindroHueco(radioInterior3 * 3, radioInterior3,
radioInterior3 * 5);
       printf("El volumen del cilindro hueco es: %g\n", volumen3);
    // Los argumentos son literales
    float volumen4 = VolumenCilindroHueco(8, 6, 15);
       printf("El volumen del cilindro hueco es: %g\n", volumen4);
}
```

16.1. Pasar parámetros de tipo por valor.

Una variable de tipo por valor contiene directamente los datos, a diferencia de una variable de tipo por referencia, que contiene una referencia a los datos. Por lo tanto, pasar una variable de tipo de valor a un método significa pasar una copia de la variable al método. Cualquier cambio en el parámetro que se produzca dentro del método no afectará a los datos originales almacenados en la variable. Si se desea que el método llamado pueda cambiar el valor del parámetro, deberá pasarlo por referencia, utilizando el operador &.

El siguiente ejemplo ilustra el paso por valor de parámetros de tipo por valor. La variable n se pasa por valor al método *Cuadrado*. Cualquier cambio que se produzca dentro del método no afectará al valor original de la variable.

```
#include <stdio.h>

#define float PI = 3.1716f

void Cuadrado(int);
```

```
main()
  {
    int n = 5;
    printf("El valor antes de llamar al método es: %d\n", n);
    Cuadrado(n);
    printf("El valor después de llamar al metodo es: %D\n ", n);
   }

void Cuadrado(int x)
  {
    x *= x;
    printf("El valor dentro del método es: %d\n", x);
  }
```

Salida del programa:

```
El valor antes de llamar al m├®etodo es: 5
El valor dentro al metodo es: 25
El valor después de llamar la al metodo es: 5
```

La variable n, que es un tipo por valor, contiene su dato, el valor 5. Cuando se invoca *Cuadrado*, el contenido de n se copia en el parámetro x, el cual se eleva al cuadrado dentro del método. Sin embargo, en main, el valor de n es el mismo, antes y después de llamar al método *Cuadrado*. De hecho, el cambio que se produce dentro del método sólo afecta a la variable local x. Nótese además que en los textos entre comillas, los caracteres especiales, (como las vocales acentuadas, no aparecen de igual forma en pantalla)

16.2. Parámetros por referencia.

El operador & produce argumentos que se van a pasar por referencia. El efecto es que cualquier cambio que se realice en el parámetro dentro del método se reflejará en esa variable cuando se devuelva el control al método de llamada.

```
#include <stdio.h>

void Cuadrado(int*);

main()
  {
    int n = 5;
    printf("El valor antes de llamar al método es: %d\n", n);
    Cuadrado(&n);
    printf("El valor después de llamar al metodo es: %d\n ", n);
   }

void Cuadrado(int *x)
  {
    int z = *x;
    z *= z;
    *x = z;
```

```
    printf("El valor dentro del metodo es: %d\n", z);
  }
```

Salida del programa:

```
El valor antes de llamar la función es: 5
El valor dentro la función es: 25
El valor después de llamar la función es: 25
```

En este ejemplo, no se pasa el valor de n, sino una referencia a n. El parámetro x no es de tipo **int** es una referencia a un valor de tipo **int**, en este caso, una referencia a n. Por lo tanto, cuando x se eleva al cuadrado dentro del método, lo que realmente se eleva al cuadrado es la variable a la que x hace referencia, es decir, n. Note que para pasar un argumento a un parámetro por referencia primero debe estar inicializado y luego al invocar el método se le debe anteponer el operador &.

17. Valores de retorno.

Los métodos pueden devolver un valor al llamador. Si el tipo de valor devuelto (el que aparece antes del nombre del método) no es **void**, el método puede devolver el valor mediante la palabra clave **return**. Una instrucción con la palabra clave **return** seguida de un valor que se corresponda con el tipo de valor devuelto devolverá ese valor al llamador del método. La palabra clave **return** también detiene la ejecución del método. Si el tipo de valor devuelto es **void**, una instrucción **return** sin ningún valor sigue siendo útil para detener la ejecución del método. Sin la palabra clave **return**, el método detendrá la ejecución cuando llegue al fin del bloque de código. Es necesario que los métodos con un tipo de valor devuelto no nulo utilicen la palabra clave **return** para devolver un valor. Si no se especifica un tipo de retorno, ni se utiliza la palabra reservada **void** se asume que el método devuelve un tipo int.

Para emplear un valor devuelto desde una función, la función que realiza la llamada puede utilizar la propia llamada a la otra función en cualquier parte donde pueda aparecer un valor del mismo tipo. El valor devuelto también se puede asignar a una variable. Los dos ejemplos de código siguientes logran el mismo objetivo:

```
main()
{
    int result = AddTwoNumbers(1, 2);
    result = SquareANumber(result);
    printf("El resultado es %d\n", result);
}

main()
{
    int result = SquareANumber(AddTwoNumbers(1, 2));
    printf("El resultado es %d\n", result);
}
```

El uso de una variable local, en este caso **result**, para almacenar un valor es opcional. Puede ayudar a la legibilidad del código o puede ser necesaria si necesita almacenar el valor original del argumento para todo el ámbito del método.

18. Variables globales y locales.

Las variables globales son aquellas que se definen en un programa fuera de todo bloque, por ello, son accesibles desde su definición, hasta el final de fichero del programa. Por el contrario, las variables que se declaran dentro de un método se conocen como variables internas o locales porque solo existen en el ámbito del método. En otras palabras, una vez que el método devuelva el control al invocador, todas las variables locales dejan de existir.

Las variables en C tienen asociada una clase de almacenamiento. Dicha clase determina su accesibilidad y existencia. Para alterar la accesibilidad y existencia de una variable, se utilizan los calificadores auto, register, static y extern. Los calificadores auto y register se utilizan solamente con variables locales, mientras que extern solo puede ser aplicado a variables globales. En cambio, el calificador static puede ser usado en ambos tipos de variables.

Las funciones también pueden ser calificadas como static y extern. Si una función es calificada como extern, puede ser accesible desde todos los ficheros fuente que componen un programa. En cambio, si una función es calificada static, solamente es accesible en el fichero donde está definida.

19. Seudocódigo para funciones o métodos.

Una función se identifica mediante su nombre. De la misma manera que cuando escribimos un algoritmo comenzamos poniendo: inicio y al final fin, debemos hacer lo mismo para una función. Esto nos dirá donde comienza y donde termina la función. La sintaxis es:

FUNCIÓN nombre_funcion (parámetros)

<instrucciones>

<instrucciones>

FIN FUNCIÓN

Todas las funciones devuelven un sólo valor. Siempre debemos indicar a la función mediante una instrucción, que devuelva el valor al algoritmo principal (recordemos que la función será llamada desde un algoritmo). Esto se debe hacer en el cuerpo de la función cuando se tenga el resultado.

nombre_funcion ← resultado.

A continuación se muestra el ejemplo de la función suma que vimos anteriormente pero esta vez expresada en seudocódigo.

```
FUNCIÓN  Suma(par1, par2)
    Suma ← par1 + par2;
FIN FUNCIÓN
INICIO
   LEER operando1
   LEER operando2
  resultadoSuma ← Suma (operando1,operando2)
   IMPRIMIR "La suma es: ", resultadoSuma
FIN
```

Note la manera en que se representa la llamada a una función:

resultadoSuma ← Suma (operando1,operando2) .

20. Ejemplo de modularidad: volumen del cilindro hueco.

Retomemos ahora el ejemplo del cilindro hueco y utilicemos exactamente el mismo algoritmo pero siguiendo los principios de programación que hemos visto en esta conferencia.

```
#include <stdio.h>
#define PI 3.1416f

float AreaBaseCilindro(float);
float VolumenCilindroHueco(float, float, float);

main()
{
	float radioExterior = 0;
	float radioInterior = 0;
	float longitud = 0;
	printf("Entre el radio exterior: \n");
	scanf("%g", &radioExterior);
	printf ("Entre el radio interior: \n");
	scanf("%g", &radioInterior);
	printf ("Entre la longitud: \n");
	scanf("%g", &longitud);
	float volumen = VolumenCilindroHueco(radioExterior, radioInterior, longitud);
	printf("El volumen del cilindro hueco es: %g\n", volumen);
}

float AreaBaseCilindro(float Radio)
{
	return PI * (Radio * Radio);
}

float VolumenCilindroHueco(float radioExterior, float radioInterior, float longitud)
{
	float volumen = 0;
	float areaBaseExterior = AreaBaseCilindro(radioExterior);
```

```
        float areaBaseInterior = AreaBaseCilindro(radioInterior);
        float areaBase = areaBaseExterior - areaBaseInterior;
        volumen = areaBase * longitud;
        return volumen;
}
```

Como se puede observar, la función **AreaBaseCilindro** permite calcular las áreas externas e internas de la base del cilindro hueco sin necesidad de hacer los cálculos por separado. El volumen se calcula al hacer una llamada al método **VolumenCilindroHueco,** pasándole como parámetro los valores de los radios externos, internos y la longitud del cilindro hueco.

21. Conclusiones

En esta conferencia se han estudiado algunos elementos del lenguaje de programación C. Al culminar esta actividad el nuevo programador deberá dominar los conceptos de identificadores, palabras reservadas, variables, constantes y literales, además de conocer todos los tipos de datos primitivos, los operadores y la estructura básica de un programa. Vencido los objetivos ya podrá dar solución a problemas sencillos que requieran solo de estructuras secuenciales.

En la próxima conferencia se verán las estructuras de control que permitirán resolver tareas más complejas cuando sea necesario evaluar una expresión y realizar acciones diferentes en base al resultado así como realizar repetitivamente acciones mientras se cumplan ciertas condiciones.

22. Estudio independiente

1. Estudiar los capítulos 1 al 3 del libro C/C++ Curso de Programación, de F.J. Ceballos. Alfaomega Group Editor, 2002.