Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Departamento de Sistemas e Informática
Escuela de Electrónica
Informática II

El lenguaje C: Elementos básicos del lenguaje

Este material debe utilizarse como **complemento** a la asistencia a las clases de teoría y práctica de la asignatura y sirve principalmente como guía general de estudio y **no como referencia exhaustiva** de los temas tratados, para los cuales debe recurrirse a la bibliografía indicada:
"**El Lenguaje de Programación C**", 2ª edición, Brian Kernighan y Dennis Ritchie, Editorial Pearson ISBN 9688802050

1. Estructura de un programa en C

En general un programa en C son muchas funciones de pequeño tamaño, y no pocas funciones de gran tamaño.

La comunicación entre las funciones es por los argumentos, valores de retorno y a través de variables externas o globales o mediante el acceso a la memoria en forma directa.

Los argumentos son siempre pasados por valor, es decir, los valores que tienen las variables o constantes usadas como argumentos en el llamado, se copian en sendas variables (argumentos) de la función llamada.

```
#include <stdio.h>
// aquí se declaran variables globales y externas
int main() {
/* Este es un comentario ignorado por el compilador */
      int index;  // declaración de una variable
      index = 13;
      printf("El valor de index is %d\n", index);
      index = 27;
      printf("El valor de index is %d\n", index);
      return 0;
}
```

C es un lenguaje que distingue entre mayúsculas y minúsculas para su escritura. En general, los programas están constituidos por una función de nombre main (que es el punto de entrada al programa) y otras funciones que son invocadas desde esta u otras.

Estructura de un programa "grande":

- Uso de archivos cabecera (.h): por lo general sólo contienen definiciones de tipos de datos, prototipos de funciones y comandos del preprocesador de C.
- Uso de varios archivos .c: por lo general con un preámbulo consistente de las definiciones de constantes, cabeceras a incluir, definición de tipos de datos, declaración de variables globales y externas (e inicialización), y una o más funciones
- División en directorios: por lo general agrupando los archivos relacionados o bajo

cierta lógica
- Uso de make y makefile: para una fácil y consistente compilación
- Uso de macros en make: típicamente usadas para guardar nombres de archivos fuente, nombres de archivos objeto, opciones del compilador o links a bibliotecas

2. Proceso de compilación

La compilación de un programa C se realiza en varias fases que normalmente son automatizadas y ocultadas por los entornos de desarrollo:

- Preprocesado: consistente en modificar el código en C según una serie de directivas de preprocesador.
- Compilación: que genera el código objeto a partir del código ya preprocesado.
- Enlazado: que une los códigos objeto de los distintos módulos y bibliotecas externas (como las bibliotecas del sistema) para generar el programa ejecutable final.

Para una compilación, estos procesos se realizan uno a continuación de otros: primero se realiza el preprocesado mediante el cual el propio código es modificado y expandido mediante el agregado de líneas de código (proveniente de otros archivos); luego el código resultante es compilado y finalmente enlazado ("linkeado")

Las directivas que pueden darse al preprocesador son, entre otras:
- **include** para la inclusión de archivos
- **define** para la definición de macros[1]
- **##** es un operador
- **# if** se usa para inclusiones condicionales

Veremos su uso con algunos ejemplos.

```
#include "nombrearchivo"
```
o
```
#include <nombrearchivo>
```
Incluye el contenido del archivo en el código. El proceso es recursivo: "nombearchivo" puede contener includes.

```
#define  nombre   texto_de_reemplazo
```
Reemplaza ***nombre*** por ***texto_de_reemplazo*** en todo el código subsiguiente.
Se usa normalmente para definir constantes.
Puede manejar argumentos. La sintaxis debe manejarse con mucho cuidado:

```
#define  PI 3.1416
#define max(A,B)    ((A)>(B)?(A):(B))
```

Si más adelante en el código en el código dice:
```
x=max(p+g,PI);
```
quedará:

[1] definición de un término o una expresión completa que será introducido en el código en reemplazo de otro.

```
x=((p+g) > (3.1416) ? (p+g) : (3.1416));
```

antes de compilar.

`#undef nombre`

Asegura que nombre no será sustituido

`##`

Concatena argumentos reales durante la sustitución

```
#define unir(inicio,fin) inicio ## fin
```

hace que:

```
A[unir(zapa,tero)];
```

pase a ser:

```
A[zapatero];
```

`# if`

Inclusión condicional:

```
#if MICRO == INTEL
        #define LIBRERIA "intel.h"
#elif MICRO == AMD
        #define LIBRERÍA "amd.h"
#else
    #define LIBRERÍA
...
/*aquí va la definicion de la librería generica*/
...
#endif
#include LIBRERIA
```

`#ifdef`
`#ifndef`

Son If especializados que testean si un *nombre* está definido

```
#ifndef LIBRERÍA
#define LIBRERÍA
....
#endif
```

3. Palabras claves del lenguaje C

El lenguaje C hace uso de las siguientes 32 palabras para la elaboración de programas:

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

4. Tipos de datos

Los tipos de datos básicos del lenguaje C y el espacio que ocupan estos en la memoria son los siguientes:

- `char` (1 byte)
- `int` (2 bytes)[2]
- `float` (4 bytes)
- `double` (8 bytes)
- `void`

Existiendo los siguientes modificadores:

```
short – long – signed – unsigned
```

que cambian la longitud o el signado del tipo al que se aplican.

5. Constantes

Para la escritura de las constantes en C en pueden utilizarse diferentes notaciones. Las que siguen son algunos ejemplos:

2323	int
5656L	long
78.56	float
89e-2	float
56.9L	double
033	octal
0xf1	hexadecimal
0xFUL	unsigned long (15 en decimal)
'n'	un carácter
'\000'	un carácter (representado en octal)
'\xhh'	un carácter (representado en hexa)

Las enumeraciones también son constantes constituidas por una lista de valores enteros constantes[3].

```
enum opcion {SI,NO,TALVEZ};  //SI vale 1, NO vale2, y así sucesivamente.
enum dia {LUNES=1, MARTES=2, ...DOMINGO='F'}
```

[2] En las máquinas y compiladores actuales los enteros ocupan de 4 bytes

[3] Es una opción al `#define`, aunque con otras connotaciones.

luego, al hacer:

```
dia k;
```

se puede hacer:

```
k=LUNES;
k=3;
```

6. Identificadores

Los identificadores se utilizan normalmente para definir los nombres de las variables, funciones, punteros[4], etc..

- El primer carácter debe ser una letra, después puede haber letras, números o guión bajo.
- Es sensible a mayúsculas y minúsculas.
- Hasta 31 caracteres.

7. Variables[5]

Deben declararse antes de utilizarse y en ese momento se "crean" en la memoria: son accesibles por su nombre.
Pueden ser:

- **externas**: se definen sólo una vez, fuera de toda función y se declaran **extern** (implícita o explícitamente) en cada función que las vaya a utilizar, siendo comunes a todas. Son inicializadas a 0 por omisión.
- **internas** (o automáticas): son declaradas dentro de una función y solo existen en ellas. No tienen una inicialización por omisión y quedan indefinidas en su valor inicial si no se las inicializa explícitamente.

además pueden ser:_

- **estáticas**: son visibles sólo dentro del archivo fuente (externas) o función (internas) en donde se declaran. Las estáticas internas mantienen su valor en sucesivos llamados a la función. Son inicializadas a 0 por omisión. Se deben inicializar con un valor o una expresión constante.
- **register**: le indica al compilador que la variable será muy usada. El compilador decidirá si será almacenada en un registro. No poseen inicialización por omisión.

8. Calificadores de tipo

Sirven para dar características especiales a los objetos (variables) que están siendo declarados:

- **const**: se puede inicializar, pero después no puede cambiarse el valor
- **volatile**: le indica al compilador que su contenido puede variar más allá del

[4] Un puntero es una variable que almacena direcciones de memoria.

[5] Debe siempre distinguirse entre el NOMBRE de la variable y la variable en sí misma. Esta última es un espacio en la memoria al cual se puede acceder, entre otras formas, mediante el nombre de la misma.

flujo del programa[6] (no son optimizables)

Ejemplos const y volatile:

```
const double e = 2.7182;
const char msg [] = "warning:";
int strlen (const char []);
const int hex = 0x80A; /* 2058 en decimal */
const int oct = 012; /* 10 en decimal */
volative int k=2;
volative unsigned char puerto1;
```

9. Arreglos y Cadenas

Los arreglos en C son de longitud fija (debe declararse su longitud), siendo su primer elementos numerado con 0 (cero).

```
int digitos [10]; //arreglo de 10 enteros. Los subíndices
                  // válidos son del 0 a 9.
int valores[] = {3,4,5,6,7}; // arreglo de longitud 5
```

En el lenguaje C no existe el tipo de dato "cadena" o "String"[7]. Para el uso de estas se utilizan arreglos de caracteres que terminan con el carácter null ("\0").

```
char patron [] = "salida";
```

es equivalente a:

```
char patron [] = {'s','a','l','i','d','a','\0'} // longitud 7
```

10. Operadores aritméticos y lógicos

El lenguaje admite los siguientes operadores aritméticos:

Operador	Nombre	Definición
*	Multiplicación	Multiplica x por y
/	División	Divide x por y
%	Modulo	Resto de x dividido y
+	Suma	Suma x más y
-	Substracción	Resta y de x
++	Incremento	++x x++
--	Decremento	--x x--
-	Negación	Multiplica x por -1

[6] Por ejemplo, por un cambio realizado al valor almacenado por medio de un dispositivo electrónico independiente del programa.

[7] No obstante, en la biblioteca estándar de C están definidas muchas funciones para la manipulación de cadenas.

+	Suma unaria	+x

que devuelven un valor acorde a la operación realizada.

Los operadores lógicos:

Operador	Ejemplo	Definición
>	x > y	1 si x es mayor que y, en caso contrario es 0
>=	x >= y	1 si x es mayor o igual a y, en caso contrario es 0
<	x < y	1 si x es menor que y, en caso contrario es 0
<=	x <= y	1 si x es menor o igual a y, en caso contrario es 0
= =	x == y	1 si x es igual que y, en caso contrario es 0
!=	x != y	1 si x no es igual que y, en caso contrario es 0
!	!x	1 si x es 0, en caso contrario es 0
&&	x && y	0 si x o y es 0, en caso contrario 1
||	x || y	0 si x e y son 0, en caso contrario 1

devuelven 1 (verdadero) o 0 (falso) según corresponda.

Por otro lado, en C, cualquier valor distinto de cero equivale a “verdadero”.

11. Operadores de acceso a datos y elementos de array

Operador	Nombre	Ejemplo	Definición
[]	Elemento de array	X[6]	7mo elemento de x
.	Selección de miembro	PORTD.B2	2do bit de PORTD
->	Selección de miembro	pStruct->x	Miembro x de la estructura apuntada por pStruct
*	Indirección	*p	Contenido de la memoria localizada en la dirección p
&	Dirección de	&x	Dirección de la variable x

12. Operadores de bits

Estos operadores trabajan sobre los bits de las variables:

Operador	Nombre	Ejemplo	Definición
~	NOT	~x	Cambia 1 por 0, y 0 por 1
&	AND	x&y	AND bit a bit de x e y
|	OR	x|y	OR bit a bit de x e y
^	XOR	x^y	XOR bit a bit de x e y
<<	Desp. Izq.	x= x<<2	Bits de x los desplaza 2 posiciones a la izquierda
>>	Desp. Der.	x=x>>2	Bits de x los desplaza 2 posiciones a la derecha

13. Sentencias de control[8]

Se muestra a continuación la sintaxis de las instrucciones que permiten controlar el flujo del programa y el establecimiento de ciclos de repetición. Estas instrucciones permiten el anidamiento de las mismas (ej.: ciclos dentro de ciclos).
En todos los casos la `expr` se refiere a cualquier expresión válida en C. Es decir, una expresión lógica, una suma, una asignación, nada, etc.

- **while**
  Repite la ejecución del bloque de código (encerrador entre { }) mientras la expresión se evalúe como verdadero (es decir, distinto de cero). Si la expresión es falsa de entrada, el bloque nunca se ejecuta.

```
while (expr){

    //bloque de código a repetir

}
```

- **do...while**
  Repite el bloque de código mientras la expresión ser verdadera. Por lo menos una vez se ejecutará el código del bloque.

```
do {

    //bloque de código a repetir

}
while(expr);
```

- **for**
  Al ejecutarse esta instrucción, primero se ejecuta *`expr1`*; a partir de allí mientras la *`expr2`* sea evaluada como verdadera, sucesivamente se ejecutará

[8] Verificar en forma precisa en la bibliografía el uso y forma de operación de estas instrucciones.

el código del bloque y ejecutará la *expr3*. El código del bloque puede entonces no ejecutarse nunca (si *expr2* es evaluada como falsa de entrada).

```
for(expr1; expr2; expr3){

    //bloque de código a repetir

}
```

- **if**
  Evalúa la expresión y si resulta verdadera ejecuta el código.

```
if (expr){
        //bloque de código
    }
```

- **if /else**
  Evalúa la expresión y si resulta verdadera ejecuta el código del bloque1. Caso contrario, ejecuta el código del bloque2.

```
if (expr){
        //bloque1
}
else {
        //bloque2
}
```

- **switch/case**
  Permite la selección múltiple trasladando la ejecución a la primera sección en la cual la expresión se evalúe igual a la constante, ejecutando a partir de allí todas las instrucciones subsiguientes[9]. Si no hay ninguna coincidencia, se ejecutan las instrucciones que siguen a `default`.

```
switch (expr)
{
    case constante1: instrucciones

    case constante2: instrucciones

    case constante3: instrucciones

. . .

    default: instrucciones
}
```

- **break**
  Al ejecutarse termina el ciclo o switch mas interno, trasladando la ejecución a la línea siguiente al mismo.

- **continue**
  Se usa sólo en ciclos y al ejecutarse provoca que se salteen las líneas del ciclo que estén “debajo” de esta instrucción, es decir, pasa a ejecutarse la siguiente iteración, con ejecución y evaluación de las expresiones de control del ciclo.

[9] Ver uso de `break` para cambiar este comportamiento.

- **goto**[10]
  Transfiere la ejecución a una línea cualquiera identificada por una etiqueta.

```
      goto final;
      ...
      //instrucciones
      ...
final:
      //instrucciones
```

14. Estructuras

Se constituyen mediante una colección de variables de distinto tipo, agrupadas bajo un nombre. Estas variables se ubicarán en forma continua en la memoria y por lo tanto el espacio que ocupe la estructura en la memoria será la suma de las variables que la constituyan.
Definida una estructura, pueden definirse variables de ese "tipo"[11]. Y a partir de la declaración de una variable de tipo estructura, puede usarse como cualquier otra variable accediendo a sus componentes mediante el operador "**.**".
Ejemplo:

```
struct point{     //defino un nombre para la estructura (es opcional)
      int x;      //defino los componentes de la estructura
      int y;      //que podrían ser de diferente tipo
};
....
struct point  pt = {3,5};     //defino la variable
printf ("%d, %d", pt.x, pt.y);// los componentes se acceden mediante el
                              // operador "."
```

Los campos o componentes de la estructura pueden ser de cualquier tipo, incluso otras estructuras (anidamiento) y arreglos.
Las estructuras también pueden formar arreglos y pasarse como argumentos a funciones como cualquier variable[12].
Ejemplo:

```
struct  message {
      int emisor;
      int receptor ;
      char datos [50];
}
struct message buffer[50];
buffer[0].emisor=0;
```

15. Uniones

Una unión es semejante a una estructura, pero los campos que se declaran ocupan todos la misma posición de memoria, es decir, están solapados, arrancando en la misma dirección y ocupando espacio según el tipo del campo. Solo se puede almacenar un campo por vez y el programador es responsable del uso debiendo asignar y recuperar coherentemente.

---

[10] En vistas a una mejor legibilidad del código debe tratar de evitarse su uso. No obstante, esta instrucción resulta de mucha utilidad cuando se desea salir del interior de ciclos anidados y otros situaciones semejantes.
[11] En C las estructuras no son formalmente un tipo de datos. En C++ sí.
[12] Para el paso de estructuras a funciones normalmente prefiere recurrirse al uso de punteros.

Cuando se define una unión solo se reserva espacio para almacenar el campo de mayor tamaño. Se usan para manejar distintos tipos de datos en una misma área de almacenamiento (*buffers*), optimizando así el uso de la memoria.

Ejemplo:

```
union ejemplo {
      int entero;        //16 bits
      char caracter;     //8 bits
} mi_var;
mi_var. entero =0;       // solo se puede incializar mediante el primer
                         //miembro
mi_var. caracter ='A';   //(como el carácter ASCII de A es 65, mi_var.entero
                         //vale 65,es decir 000000001000001)
mi_var. entero= 65;      // pero no es seguro que mi_var.caracter sea una
                         //'A'
```

16. Campo de bits

Es un conjunto de bits adyacentes, agrupados, en una palabra. Los bits se manipulan según su tipo declarado pero su longitud ha sido llevada al tamaño de uno o más bits.[13]
Ejemplo:

```
struct {
      unsigned int ctrl_compresor :1; //es un entero pero de longitud 1
      unsigned int ctrl_cinta :1;
      unsigned int ctrl_quemador :1;
} control;                          // control es una palabra de 4 bits
...
control.ctrl_compresor=1; //enciende el compresor
```

17. Creación de tipos de datos

La instrucción typedef define un tipo de datos a partir de los tipos básicos:

```
typedef unsigned char byte;
typedef int Longitud;
```

hace que byte sea de 8 bits sin signo y que Longitud sea un sinónimo de int. Luego se puede hacer:

```
Longitud k;
Byte high=255;
```

El siguiente código

```
typedef struct {
      int coor_x;
      int coor_y;
} Complejo;
```

crea el tipo Complejo que es análogo a la estructura definida en este caso.

Ejemplo:

[13] Los campos de bits son particularmente útiles en la programación de microcontroladores. Tener en cuenta que la implantación define si pueden o no existir "solapamientos".

```
typedef unsigned char byte; //crea el tipo byte
typedef union {
  byte Byte;              //campo Byte de la union
  struct {
        byte b0      :1;
        byte b1      :1;
        byte b2      :1;
        byte b3      :1;
  } Bits;                 //campo Bits de la union: se puede acceder por bits
} Control;                //crea el tipo Control
```

luego:

```
Control Z;                //declara la variable
Z.Byte= 4;
Z.Bits.b2=~Z.Bits.b2; //invierte el tercer bit
```

18. Otros aspectos del lenguaje C

Son vistos en los apuntes de la cátedra sobre punteros y entrada/salida en conjunto con el lenguaje C++, recomendándose su lectura.

Asimismo, se insiste en recurrir a la bibliografía recomendada (“El Lenguaje de Programación C”, de Kernighan & Ritchie) para el correcto estudio y comprensión de todos los temas del lenguaje.