LENGUAJE ANSI C
Guía de referencia

C.1. Elementos básicos de un programa

El lenguaje C fue desarrollado en Bell laboratories para su uso en investigación y se caracteriza por un gran número de propiedades que lo hacen ideal para usos científicos y de gestión.

Una de las grandes ventajas del lenguaje C es ser *estructurado*. Se pueden escribir bucles que tienen condiciones de entrada y salida claras y se pueden escribir funciones cuyos argumentos se verifican siempre para su completa exactitud.

Su excelente biblioteca estándar de funciones, convierten a C en uno de los mejores lenguajes de programación que los profesionales informáticos pueden utilizar.

C.2. Estructura de un programa C

Un programa típico en C se organiza en uno o más *archivos fuentes* o *módulos*. Cada archivo tiene una estructura similar con comentarios, directivas de preprocesador, declaraciones de variables y funciones y sus definiciones. Normalmente se sitúan cada grupo de funciones y variables relacionadas en un único archivo fuente. Dentro de cada archivo fuente, los componentes de un programa suelen colocarse en un determinado modo estándar. La Figura B.1 muestra la organización típica de un archivo fuente en C

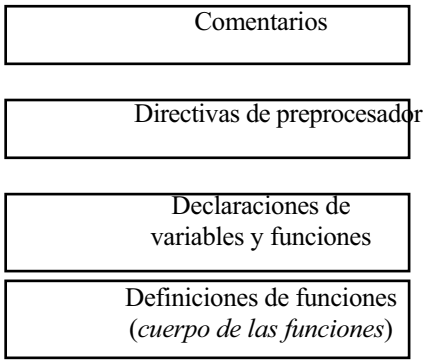

Figura B.1. Organización de un programa C

Los componentes típicos de un archivo fuente del programa son:

1. El archivo comienza con algunos comentarios que describen el propósito del módulo e información adicional tal como el nombre del autor y fecha, nombre del archivo. Los comentarios comienzan con `/*` y terminan con `*/`.
2. Ordenes al preprocesador, conocidas como *directivas del preprocesador*. Normalmente incluyen archivos de cabecera y definición de constantes.
3. Declaraciones de variables y funciones son visibles en todo el archivo. En otras palabras, los nombres de estas variables y funciones se pueden utilizaren cualquiera de las funciones de este archivo. Si se desea limitar la visibilidad de las variables y funciones *sólo* a ese módulo, ha de poner delante de sus nombres el prefijo static; por el contrario la palabra reservada extern indica que los elementos se declaran y definen en otro archivo.
4. El resto del archivo incluye definiciones de las funciones (su cuerpo). Dentro de un cuerpo de una función se pueden definir variables que son locales a la función y que sólo existe en el código de la función que se está ejecutando.

B.3. El primer programa C ANSI

```
#include <stdio.h>

main ()
{
        printf ("¿Hola mundo!");
  return 0;
}
```

B.4. Palabras reservadas ANSI C

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

Las palabras reservadas const, enum, void y volatile son nuevas en ANSI C.

B.5. Directivas del preprocesador

El *preprocesador* es la parte del compilador que realiza la primera etapa de traducción o compilación de un archivo C ANSI en instrucciones de máquina. El preprocesador procesa el archivo fuente y actúa sobre las órdenes, denominadas *directivas de preprocesador*, incluidas en el programa. Estas directivas comienzan con el signo de libra (almohadilla) #. Normalmente, el compilador invoca automáticamente al preprocesador antes de comenzar la compilación. Se puede utilizar el preprocesador de tres formas distintas para hacer sus programas más modulares, más legibles y más fáciles de personalizar:

1. Se puede utilizar la directiva #include para insertar el contenido de un archivo en su programa.
2. Mediante la directiva #define, se pueden definir macros que permiten reemplazar una cadena por otra. Se puede utilizar la directiva #define para dar nombres significativos a constantes numéricas, mejorando la legibilidad de sus archivos fuente.
3. Con directivas tales como #if, #ifdef, #else y #endif, pueden compilar sólo partes de su programa. Se puede utilizar esta característica para escribir archivos fuente con código para dos o más sistemas, pero compilar sólo aquellas partes que se aplican al sistema informático en que se compila el programa.

B.6. Archivos de cabecera

Directivas tales como #include <stdio.h> indica al compilador que lea el archivo *stdio.h* de modo que sus líneas se sitúan en la posición de la directiva. C ANSI soporta dos formatos para la directiva #include:

1. #include <stdio.h>
2. #include "demo.h"

El primer formato de #include, lee el contenido de un archivo - el archivo estándar de C, *stdio.h* . El segundo formato, visualiza el nombre del archivo encerrado entre las dobles comillas que está en el directorio actual.

B.7. Definición de macros

Una macro define un símbolo equivalente a una parte de código C y se utiliza para ello la directiva #define. Se puede representar constantes tales como PI, IVA y BUFFER.

```
#define PI 3.14159
#define IVA 16
#define BUFFER 1024
```

que toman los valores 3.14159, 16 y 1024 respectivamente. Una macro también puede aceptar un parámetro y reemplazar cada ocurrencia de ese parámetro con el valor proporcionado cuando el macro se utiliza en un programa. Por consiguiente, el código que resulta de la expansión de una macro puede cambiar dependiendo del parámetro que se utilice cuando se ejecuta la macro. Por consiguiente, el código que resulta de la expansión de una macro puede cambiar dependiendo del parámetro que se utilice cuando se ejecuta la macro. Por ejemplo, la macro siguiente acepta un parámetro y expande a una expresión diseñada para calcular el cuadrado del parámetro

```
#define cuadrado (x) ((x)*(x))
```

B.8. Comentarios

El compilador ignora los comentarios encerrados entre los símbolos /* y */.

```
/* Mi primer programa */
```

Se pueden escribir comentarios multilínea

```
/* Mi segundo programa C
escrito el día 15 de Agosto de 1985
en Carchelejo - Jaén - España */
```

Los comentarios no pueden anidarse. La línea siguiente no es legal:

```
/*Comentario /* comentario interno */ externo */
```

B.9. Tipos de datos

Los tipos de datos básicos incorporados a C son *enteros*, *reales* y *carácter.*

Tabla B.1. Tipos de datos enteros

Tipo de dato	Tamaño en bytes	Tamaño en bits	Valor mínimo	Valor máximo
signed char	1	8	-128	127
unsigned char	1	8	0	255
signed short	2	16	-32.768	32.767
unsignet short	2	16	0	65.535
signed int	2	16	-32.768	32.767
unsigned int	2	16	0	65.535
signed long	4	32	-2.147.483.648	2.147.483.647
unsigned long	4	32	0	4.294.967.295

El tipo char se utiliza para representar caracteres o valores integrales. Las constantes de tipo char pueden ser caracteres encerrados entre comillas (´A´, ´b´, ´p ´). caracteres no imprimibles (tabulación, avance de página...) se pueden representar con secuencias de escape (´\t´, ´\f´)

Tabla B.2. Secuencias de escape

Carácter	Significado	Código ASCII
\a	Carácter de alerta (timbre)	7
\b	Retroceso de espacio	8
\f	Avance de página	12
\h	Nueva línea	10
\r	Retorno de carro	13
\t	Tabulación (horizontal)	9
\v	Tabulación (vertical	11
\\	Barra inclinada	92
\?	Signo de interrogación	63
\´	Comilla	39
\´´	Doble comilla	34
\nnn	Número octal	-
\xnn	Número hexadecimal	-
´\0´	Carácter nulo (terminación de cadena)	-

Tipos de datos en como flotante

Tipo de dato	Tamaño en bytes	Tamaño en bits	Valor mínimo	Valor máximo
float	4	32	3.4E-38	3.4E+38
double	8	64	1.7E-308	1.7E+308
long double	10	80	3.4E-4932	1.1E+4932

Todos los números sin un punto decimal en programas C se tratan como enteros y todos los números con un punto decimal se consideran reales de coma flotante de doble precisión. Si se desea representar números en base 16 (hexadecimal) o en base 8 (octal), se precede al número con el carácter ´OX´ para hexadecimal y ´0´ para octal. Si se desea especificar que un valor entero se almacena como un entero largo se debe seguir con una ´´L´´

```
025          /* octal 25 o decimal 21 */
0x25   /* hexadecimal 25 o decimal 37 */
250L /* entero largo 250 */
```

B.10 .Variables

Todas las variables en C se declaran o definen antes de que sean utilizadas. Una *declaración* indica el tipo de una variable.

Si la declaración produce también almacenamiento (se inicia), entonces en una *definición.*

B.10.1. Nombres de variables en C

Los nombres de variables en C constan de letras, números y carácter subrayado. Pueden ser mayúsculas o minúsculas o una mezcla de tamaños. El tamaño de la letra es significativo. las variables siguientes son *todas diferentes*.

```
Temperatura   TEMPERATURA          temperatura
```

A veces se utilizan caracteres subrayados y mezcla de mayúsculas y minúsculas para aumentar la legibilidad:

```
Dia-de-Semana          DiaDeSemana      Nombre-ciudad      PagaMes

int x;                                  //declara x variable entera
char nombre, conforme;          //declara nombre, conforme de tipo char
```

```
int x =, no = 0;                    //definen las variables x y no
float total = 42.125                //define la variable total
```

Se pueden declarar variables múltiples del mismo tipo de dos formas: Así una declaración

```
int v1; int v2; int v3; int v4;
```

o bien

```
int v1;
int v2;
int v3;
int v4;
```

pudiendose declarar también de la forma siguiente:

```
int v1, int v2, int v3, int v4;
```

C, no soporta tipos de datos lógicos, pero mediante enteros se pueden representar: 0, significa *falso*; distinto de cero, significa *verdadero* (cierto).

La palabra reservada const permite definir determinadas variables con valores constantes, que no se pueden modificar. Así, si se declara

```
const int z = 4350
```

y si se trata de modificar su valor

```
z = 3475
```

el compilador emite un mensaje de error similar a "Cannot modify a const object in function main" ("No se puede modificar un objeto const en la función main"). Las variables declaradas como const pueden recibir valores iniciales, pero no pueden modificarse su valor con otras sentencias.

B.10.2. Variables tipo char

Las variables de tipo char (carácter) pueden almacenar caracteres individuales. Por ejemplo la definición

```
char car = ´M´;
```

declara una variable car y le asigna el valor ASCII del carácter M. El compilador convierte la constante carácter ´M´ en un valor entero (int), igual al código ASCII de ´M´ que se almacena a continuación en el byte reservado para car.

Dado que los caracteres literales se almacenan internamente como valores int, se puede cambiar la línea.

```
char car;
```

por

```
int car;
```

y el programa funcionará correctamente

B.10.3. Constantes de cadena

Las cadenas de caracteres constan de cero o más caracteres separados por dobles comillas. La cadena se almacena en memoria como una serie de valores ASCII de tipo char de un solo byte y se termina con u byte cero, que se llama carácter nulo

```
´Sierra Mágina en Jaén"
```

Además de los caracteres que son imprimibles, se pueden guardar en constantes cadena, códigos de escape, símbolos especiales que representan códigos de control y otros valores ASCII no imprimibles. Los códigos de escape se representan en la Tabla como un carácter único, almacenado internamente como un valor entero y compuesto de una barra inclinada seguida por una letra, signo de puntuación o dígitos octales o hexadecimales. Por ejemplo, la declaración

```
char c = ´\n´;
```

asigna el símbolo nueva línea a la variable C. En los PC, cuando se envía un carácter \n a un dispositivo de salida, o cuando se escribe \n en un archivo de texto, el símbolo nueva línea se convierte en un retorno de carro y un avance de línea.

B.10.4. Tipos enumerados

El tipo enum es una *"lista ordenada"* de elementos como constantes enteras. A menos que se indique lo contrario, el primer miembro de un conjunto enumerado de valores toma el valor 0, pero se pueden especificar valores. La declaración:

```
enum diasSemana {Lunes, Martes, Miercoles, Jueves, Viernes,
                                    Sabado, Domingo };
```

significa que Lunes = 0, martes = 1 etc. Sin embargo, sise hace Viernes = 10, entonces Lunes sigue siendo 0, Martes es igual a 2 etc.; pero ahora Viernes = 11, Sabado = 12, etc.

Un tipo enumerado se puede utilizar para declarar una variable

```
enum diasSemana Laborable;
```

y a continuación utilizarla con

```
Laborable = Jueves;
```

o bien

```
Laborable = Sabado;
if (Laborable >= Viernes)
    printf ( "Hoy no es laborable \n",Laborable);
```

B.11. Expresiones y operadores

Las expresiones son operaciones que realiza el programa.

```
a+b+c;
```

Tabla B.3. Operadores aritméticos

Operador	Descripción	Ejemplo
*	Multiplicación	(a*b)
/	División	(a/b)
+	Suma	(a+b)
-	Resta	(a-b)
%	Módulo	(a%b)

Tabla B.4. Operadores relacionales

Operador	Descripción	Ejemplo
<	Menor que	(a<b)
<=	Menor que o igual	(a <= b)

>	Mayor que	(a > b)
>=	Mayor o igual que	(a >= b)
= =	Igual	(a = = b)
!=	No igual	(a != b)

Tabla B.6. Operadores de incremento y decremento

Operador	Descripción	Ejemplo
++	Incremento en i	++i, i++
--	Decremento en i	--j, j--

Ejemplos

```
i = i+i;          //Sumar uno a i
i++;                        //Igual que anterior

i = i-i:          //Resta uno a i
i--;                        //Igual que anterior
```

Tabla B.7. Operadores de manipulación de bits

Operador	Descripción	Ejemplo
&	AND bit a bit	C = A&B;
|	OR inclusiva bit a bit	C = A|B;
^	OR exclusiva bit a bit	C = A^B;
<<	Desplazar bits a izquierda	C = A<<B;
>>	Desplazar bits a derecha	C = A>>B;
-	Complemento a uno	C = -B

B.11.1. Operadores de asignación

Los operadores de asignación son binarios y combinaciones de operadores y del signo = utilizado para abreviar expresiones:

```
A = B                    /* asigna el valor de B a A
C = (A=B)      /* C y A son iguales a B
C = A =B       /* asigna B a A y a C
```

A = A + 45;	*equivale a*	A + = 45;

El compilador puede generar código más eficiente, recurriendo a operadores de asignación compuestos del tipo * =, + = etc. cada operador compuesto (*op*) reduce la expresión en pseudocódigo:

O = a *op* b

a la forma abreviada

a *op* = b;

Tabla B.8. Operadores de asignación

Operador	Descripción	Ejemplo

=	Operación de asignación simple		a = b;
* =	z * = 10;	*equivale a*	z = z * 10;
/ =	z / = 5;	*equivale a*	z = z / 5;
% =	z % = 2;	*equivale a*	z = z % 2;
+ =	z + = 4;	*equivale a*	z = z + 4;
- =	z - = 5;	*equivale a*	z = z - 5;
<< =	z << = 3;	*equivale a*	z = z << 3;
>> =	z >> = 4;	*equivale a*	z = z >> 3;
& =	z & = j;	*equivale a*	z = z $ j;
^ =	z ^ = j;	*equivale a*	z = z ^j;
| =	z | = j;	*equivale a*	z = z | j;

B.11.2. Operador serie

El operador en serie, la coma, indica una serie de sentencias ejecutadas de izquierda a derecha. Se utiliza normalmente en bucles for. Por ejemplo:

```
for (cuenta=1; cuenta<100; ++cuenta, ++lineasporpagina);
```

produce el incremento de la variable cuenta y de la variable lineasporpagina cada vez que se ejecuta el bucle (se realiza una iteración).

B.11.3. Prioridad (precedencia) de operadores

Las expresiones C constan de diversos operandos y operadores. En expresiones complejas, las subexpresiones con operadores de prioridad (precedencia) más alta se evalúan antes que las subexpresiones con operadores de menor prioridad.

Tabla B.9. Orden de evaluación y prioridad de operadores (asociatividad)

Nivel	Operadores	Orden de evaluación
1	( ) . Ψβ >	izquierda-derecha
2	* & ! ~ ++ - - + - *(conversión de tipo sizeof)*	derecha-izquierda
3	* / %	izquierda-derecha
4	+ -	izquierda-derecha
5	<< >>	izquierda-derecha
6	< <= > >=	izquierda-derecha
7	= = !=	izquierda-derecha
8	&	izquierda-derecha
9	^	izquierda-derecha
10	|	izquierda-derecha
11	&&	izquierda-derecha
12	||	izquierda-derecha
13	?:	derecha-izquierda
14	= *= /= += -= %= <<= >>= &= ^= |=	derecha-izquierda
15	,	izquierda-derecha

B.12. Funciones de entrada y salida

Las funciones printf() y scanf() permiten comunicarse con un programa. Se denominan funciones de E/S. printf() es una función de salida (S) y scanf() es una función de entrada y ambas utilizan una cadena de control y una lista de argumentos.

B.12.1 printf

La función printf escribe en el dispositivo de salida, los argumentos de la lista de argumentos. Requiere el archivo de cabecera *stdio.h*. La salida de printf se realiza con formato y su formato consta de una cadena de control y una lista de variables.

printf (cadena de control Ψitem1, item2,...itemβ);

El primer argumento es la *cadena de control* (o formato); determina el formato de escritura de los datos. Los argumentos restantes son los datos o variables de datos a escribir

printf ("Esto es una prueba %d\n", prueba);

La cadena de control tiene tres componentes: texto, identificadores y secuencias de escape. Se puede utilizar cualquier texto y cualquier número de secuencias de escape. El número de identificadores ha de corresponder con el número de variables o valores a escribir.

Los identificadores de la cadena de formato determinan como se escriben cada uno de los argumentos.

printf ("Mi pueblo favorito es Cazorla%s", msg);

cada identificador comienza con un signo porcentaje (%) y un código que indica el formato de salida de la variable.

Tabla B.10. Códigos de identificadores

Identificador	Formato
% d	Entero decimal
% c	Carácter simple
% s	Cadena de caracteres
% f	Coma flotante (decimal)
% e	Coma flotante (notación exponencial)
% g	Usa el %f o el %e más corto
% u	Entero decimal sin signo
% o	Entero octal sin signo
% x	Entero hexadecimal sin signo

Las secuencias de escape son las indicadas en la tabla

```
printf ("Mi flor favorita es la %s\n", msg);
printf ("La temperatura es % f grados centigrados \n",
  centigrados);
```

Ejemplo B.1.

```
#include <stdio.h>
#define PI 3.141593
#define SIERRA "Sierra Mágina"
int main (void)
{
        printf ("El valor de pi es % f.\n", PI);
        printf ("/%25/ \n", SIERRA);
        printf ("/%225/ \n", SIERRA);
        return 0;
```

```
}
```

La salida producida al ejecutar el programa es

```
El valor de PI es 3.141593.
/Sierra Magina/
/                SierraMagina/
```

B.12.2. scanf

La función scanf() es la función de entrada con formato. Esta función se puede utilizar para introducir números con formato de máquina, caracteres o cadena de caracteres, a un programa

```
scanf ("%f", &fahrenheit);
```

El formato general de la función scanf() es una cadena de formato y uno o más variables de entrada. La cadena de control consta sólo de identificadores.

Tabla B.11. Identificadores de formato de scanf

Identificador	Formato
% d	Entero decimal
% c	Carácter simple
% s	Cadena de caracteres
% f	Coma flotante
% e	Coma flotante
% u	Entero decimal sin signo
% o	Entero octal sin signo
% x	Entero hexadecimal sin signo
& h	Entero corto

Un ejemplo típico de uso de scanf es

```
printf ("Introduzca ciudad y provincia : ");
scanf ("%s %s, ciudad, provincia);
```

Otros ejemplos de uso de scanf

```
scanf ("%d", &cuenta);
scanf ("%205", direccion)
scanf ("%d%d", &r, &c);
scanf ("%d*c%d", &x, &y);
```

B.13. Sentencias de control

Una sentencia consta de palabras reservadas, expresiones y otras sentencias. cada sentencia termina con un punto y coma (;).

Un tipo especial de sentencia, la *sentencia compuesta* o *bloque*, es un grupo de sentencias encerradas entre llaves ({...}). El cuerpo de una función es una sentencia compuesta. Una sentencia compuesta puede tener variables locales.

B.13.1. Sentencia if

1. if (*expresión*) *sentencia*;

o bien

if (*expresión*) *sentencia;*

2. Si la sentencia es *compuesta*

```
if (expresión) {          o bien          if (expresión)

        {
  sentencia1;                                          sentencia1;
  sentencia2;                                          sentencia2;
  ...
...
}
}
```

3. if (*expresión = = valor*)
   sentencia;

4. if (*expresión != 0*) *equivale a* if (*expresión*)
   sentencia;

Nota: (expresión != 0) y (*expresión*) son equivalentes, ya que cualquier valor distinto de cero representa *cierto*.

B.13.2. Sentencia if-else

1. if (*expresión*)
   sentencia1;
else
   sentencia2;

2. if (*expresión*) { *o bien* if (*expresión*)

```
                {
  sentencia1;
sentencia1;
  sentencia2;
sentencia2;
  ...                                                  }
} else
else

        {
  sentencia3;
sentencia3;
  sentencia4;
sentencia4;
  ...
}
}
```

Sentencias* if *anidadas

1. if (*expresión1*)
   sentencia1;
else if (expresión2)
   sentencia2;
else
   sentencia3;

2. if (*expresión1*)
   sentencia1;

```
    else if (expresión2)
      sentencia2;
    else if (expresión3)
      sentencia3;
    else if (expresiónN)
      sentenciaN;
    else
                                                    /* opcional */
          SentenciaPorOmision;      /* opcional */
```

B.13.3. Expresión condicional (?:)

Expresión condicional es una simplificación de una sentencia if-else. Su sintaxis es:

```
    expresión1 ?   expresión2 : espresión3;
```

que equivale a

```
    if (expresión1)
       expresión2;
    else
       expresión3;
```

Ejemplo B.2

```
if (opcion == ´S´)  equivale a          Premio=(opción == ´S´)? 1000 = 0
     premio = 1000
else
  premio = 0
```

B.13.4. Sentencia switch

La sentencia switch realiza una bifurcación múltiple, dependiendo del valor de una expresión

```
switch (expresión) {
    case valor1:
          sentencia1;          /*se ejecuta si expresión igual a 1 */
          break;                  /* salida de sentencia switch      */
          case valor2:
                sentencia2;
          break;
    case valor3:
                sentencia3;
          break;
    ...
    default:
          sentenciaporomision         /* se ejecuta si ningún valor coincide con expresión */
}
```

Ejemplo B.3.

```
    switch (op)
    {
          case ´a´:
            func1();
            break;
          case ´b´:
            func2();
            break;
```

```
        case ´H´:
          printf (“Hola \n”);
        default:
          printf (“Salida \n”);
```

B.13.5. Sentencia while

```
while (expresión)          o bien          while (expresión) {
    sentencia;                                                          sentencia1;

sentencia2;

...

}
```

B.13.6. Sentencia do-while

```
do {                                                                    o bien                  do {
   sentencia;
   sentencia1;
        ...
        sentencia2;
} while (expresión);                                                                                ...

        }           while (expresión)
```

Las sentencias do-while monosentencias se pueden escribir también así:

do *sentencia*; while (*expresión*);

B.13.7. Sentencia for

1. for (*expresión1*; *expresión2*; *expresión3*) {
sentencia;
}

2. for (*expresión1; expresión2; expresión3*) *sentencia*;

La sentencia for equivale a
expresión1;
while (*expresión2*) {
sentencia;
expresión3;
}

Ejemplo B.4.

a.
```
for (i = i; i <= 100; i++)
        printf (“i = = %d\n”, i);
```

b.
```
for (i = 0, suma = 0; i <= 100; suma + = i, i++);
```

sentencia nula

B.13.8. Sentencia nula

La *sentencia nula* representada por un punto y coma no hace nada. Se utilizan sentencias nulas en bucles, cuando todo el proceso se hace en las expresiones del bucle en lugar del cuerpo. Por ejemplo, localizar el byte cero que marca el final de una cadena:

```
char cad Ψ80β= “Prueba”;
int i;

for (i = o; cad Ψiβ!= ´\0´; i++)
        ;           /* sentencia nula
```

B.13.9. Sentencia break

La sentencia break se utiliza para salir incondicionalmente de un bucle for, while, do-while o de una sección case de una sentencia switch.

```
for (;;) {
        ...
                if (expresion)
                break;
}
```

B.13.10. Sentencia continue

La sentencia continue salta en el interior del bucle hasta el principio del bucle para proseguir con la *ejecución*, dejando sin ejecutar las líneas restantes después de la sentencia continue hasta el final del bucle. continue es similar a la sentencia break.

```
while (expresion1) {
                ...
                if (expresion2)
                 continue:
                ...
}
```

B.13.11. Sentencia goto

Una sentencia goto dirige el programa a una sentencia específica que tiene etiqueta utilizada en dicha sentencia goto. Las etiquetas deben terminar con un símbolo dos puntos.

```
salto:
        ...
        if (expresion) goto salto;
```

B.14. Funciones

Las funciones son los bloques de construcción de programas C. Una *función* es una colección de declaraciones y sentencias. Cada programa C tiene al menos una función: la función main. Esta es la función donde comienza la ejecución de un programa C. La biblioteca ANSI C contiene gran cantidad de funciones estándar.

B.14.2. Sentencia return

La sentencia return detiene la ejecución de la función actual y devuelve al control a la función llamadora. La sintaxis es:

return *expresion*

en donde el valor *de expresión* se devuelve como valor de la función.

```
#include <stdio.h>

main()
{
        printf ("Sierra de Cazorla y \n");
        printf ("Sierra Magina \n")N
        printf ("son dos hermosas sierras andaluzas \n");
        return 0;
}
```

B.14.1. Prototipos de funciones

En ANSI C se debe declarar una función antes de utilizarla. La declaración de la función indica al compilador el tipo de valor que devuelve la función y el número y tipos de argumentos que acepta.

El *prototipo de una función* es el nombre de la función, la lista de sus argumentos y el tipo de dato que devuelve.

```
int SeleccionarMenu(void);
double Area(int x, int y);
void salir(int estado);
```

B.14.3. El tipo void

Si una función no devuelve nada, ni acepta ningún parámetro, ANSI C ha incorporado el tipo de dato void, que es útil para declarar funciones que no devuelven nada y para describir punteros que pueden apuntar a cualquier tipo de dato.

```
void exit (int estado);
void CuentaArriba (void);
void CuentaAbajo (void);
```

Errores típicos de funciones

1. *Ningún retorno de valor.* La función carece de una sentencia return. Si una función termina sin ejecutar return, devuelve un valor impredecible, que puede producir errores serios.
2. *Retornos omitidos*. Si hay funciones que tienen sentencias if, hay que asegurarse de que existe una sentencia return por cada camino de salida posible.
3. *Ausencia de prototipos*. Las funciones que carecen de prototipos se consideran devuelven int, incluso aunque estén definidas para devolver valores de otro tipo. Como regla general, declarar prototipos para todas las funciones.
4. *Efectos laterales*. Este problema se produce normalmente por una función que cambia el valor de una o más variables globales.

B.14.4. Detener un programa con exit

Un medio para terminar un programa sin mensajes de error en el compilador es utilizar la sentencia

```
return valor;
```

donde *valor* es cualquier expresión entera.

Otro medio para detener un programa es llamar a exit. La ejecución de la sentencia

```
exit(o);
```

termina el programa inmediatamente y cierra todos los archivos abiertos.

B.15. Estructuras de datos

Los tipos complejos o estructuras de datos en C son: *arrays, estructuras, uniones, cadenas* y *campos de bits*.

B.15.1. Arrays

Todos los arrays en C comienzan con el índice 0.

```
int notas[25];                  /*declara array de 25 elementos*/
float lista[10][25];  /* declara array de 10 por 25
                                                    dimensiones */
```

El número de elementos de un array se puede determinar dividiendo el tamaño del array completo por el tamaño de uno de los elementos

```
noelementos = sizeof(lista) /sizeof (lista[0])
```

Un *array estático* se puede iniciar cuando se declara con:

```
static char nombre[]= "Sierra Magina";
```

Un *array auto* se puede iniciar sólo con una expresión constante en ANSI C.

```
int listamenor [2][2]= {{25, 4}, {100, 75}};
int digitos[]= {0,1, 2, 3, 4, 5, 6, 7, 8, 9, };
```

Un *array de punteros* a caracteres se puede iniciar con

```
char*colores[]= {"verde", "rojo", "amarillo", "rosa", azul"};
```

B.16. Cadenas

Las *cadenas* son simplemente arrays de caracteres. las cadenas se terminan siempre con un valor nulo ('\0')

```
char cadena[80];                /* declara una cadena de 80 caracteres */
char mensaje[]= "Carchelejo está en Sierra Mágina";
char frutas[][10]= {"naranja", "platano", manzana" };
```

B.17. Estructuras

Las **estructuras** son colecciones de datos, normalmente de tipos diferentes que actúan como un todo. Puede contener tipos de datos simples (carácter, float, array y enumerado) o compuestos (estructuras, arrays o uniones)

```
struct coordenada {
        int x;
        int y;
};

struct signatura {
        char titulo [40];
        char autor [30];
        int paginas;
        int anyopubli;
};
```

Una variable de tipo estructura se puede declarar así:

```
struct coordenada punto;0
        struct signatura librosinfantiles;
```

Para asignar valores a los miembros de una estructura se utiliza la notación punto (.)

```
punto.x = 12;
punto.y = 15;
```

Existe otro modo de declarar estructuras y variables estructuras.

```
struct coordenada {
                int x;
                int y;
} punto;
```

Para evitar tener que escribir struct cada vez que se declara la estructura, se puede usar typedef en la declaración:

```
typedef struct coordenada {
                int x;
                int y;
}Coordenadas;
```

y a continuación se puede declarar la variable Punto:

```
Coordenadas punto;
```

y utilizar la notación punto (.)

```
punto.x = 12
punto.y = 15;
```

B.18. Uniones

Las uniones son casi idénticas a las estructuras en su sintaxis. Las uniones proporcionan un medio de almacenar más de un tipo en una posición de memoria. Se define un tipo unión

```
union tipodemo {
                short x;
                long l;
                float f;
};
```

y se declara una variable con

```
union tipodemo demo;
```

y se puede utilizar

```
demo.x = 345;
```

o bien

```
demo.y = 324567;
```

La unión anterior se puede iniciar así:

```
union tipodemo { short x; long l; float f; } = {75};
```

B.19. Campos de Bits

Los *campos de bits* se utilizan con frecuencia para poner enteros en espacios más pequeños de los que el compilador pueda normalmente utilizar y son, por consiguiente, dependientes de la implementación. Una estructura de campos de bits especifica el número de bits que cada miembro ocupa.

Una declaración de un estructura Persona

```
struct persona {
        unsigned edad;          /* 0::99 */
        unsigned sexo;          /* 0 = varon, 1 = hembra */
        unsigned hijos;         /* 0 .. 15 */
};
```

y de una estructura de campos de bits

```
struct persona {
        unsigned edad: 7;       /* 0..1127 */
        unsigned sexo: 1;       /0 = varon, 1 = hembra */
        unsigned hijos: 4;      /* 0..15 */
}
```

B.20. Punteros

El **puntero** es un tipo de dato especial que contiene la dirección de otra variable. Un puntero se declara utilizando el asterisco (*) delante de un nombre de variable

```
float *longitudOnda;        /* puntero a datos float */
char *indice;               /* puntero a datos char  */
int *p;                     /* puntero a datos int  */
```

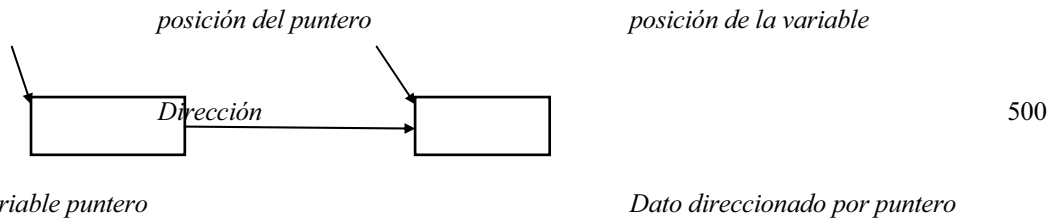

Figura B.2. Un puntero es una variable que contiene una dirección

B.20.1. Declaración e indirección de punteros

```
int m;          /* una variable entera un */
int *p          /* un puntero p apunta a un valor entero */
p = &i;         /*se asigna a p la dirección de la variable m*/
```

El operador de indirección (&) se utiliza para acceder al valor de la dirección contenida en un puntero.

```
float *longitudOnda;
longitudOnda = &cable1;
*longitudOnda = 40.5;
```

B.20.2. Punteros nulos y void

Un puntero nulo apunta a ninguna parte específica; es decir, no direcciona a ningún dato válido en memoria:

```
fp = NULL;          /* asigna Nulo a fp */
fp = 0;                                 /* asigna Nulo a fp */

if (fp != NULL)     /*  el programa verifica si el puntero es válido */
```

Al igual que una función void que no devuelve ningún valor, un puntero void apunta a un tipo de dato no especificado

```
void *noapuntafijo;
```

B.20.3. Punteros a valores

Para utilizar un puntero que apunte a un valor, se utiliza el operador de indirección (*) delante de la variable puntero

```
double *total;
*total = 34750.75;
```

El operador de dirección (&) asigna el puntero a la dirección de la variable

```
double *total;
double sctotal;
total = &ctotal;
```

B.20.4. Punteros y arrays

Si un programa declara

```
float *punterolista;
float listafloat [30];
```

entonces a punterolista se puede asignar la dirección del principio de listafloat (el primer elemento).

```
punterolista = listafloat;
```

o bien

```
punterolista = &listafloat [0];
listafloat [4]       es lo mismo que   *(listafloat + 4)
listafloat [2]       es lo mismo que   *(listafloat + 2)
```

B.20.5. Punteros a punteros

Los punteros pueden apuntar a otros punteros (*doble indirección*)

```
char **lista;       /* se declara un puntero a otro puntero char */
```

equivale a

```
char *lista [];
```

y

```
char ****ptr   /* un puntero a un puntero a un puntero a char */
```

B.20.6. Punteros a funciones

```
float *ptr1;                          /* ptr apunta a un valor float */
float *mifun(void);       /* la función mifunc devuelve un
                                                    puntero a un valor float */
```

La declaración

```
int (*ptrafuncion((void);
```

crea un puntero a una función que devuelve un entero y la declaración

```
float (*ptrafuncion1)(int x, int y);
```

declara ptrafuncion1 como un puntero a una función que devuelve un valor float y requiere dos argumentos enteros.

A.21 PREPROCESADOR DE C

El preprocesador de C es un conjunto de sentencias específicas, denominadas *directivas*, que se ejecutan al comenzar el proceso de compilación.

Una directiva comienza con el símbolo # como primer carácter que indica al preprocesador que ha de ejecutarse una acción específica.

A.21.1 Directiva `#define`

Se utiliza para asociar identificadores con una secuencia de caracteres, estos pueden ser una palabra reservada, una constante, una sentencia o una expresión. En el caso de que el identificador represente sentencias o expresiones se denomina *macro*.

Sintaxis **#define** <identificador>[(parámetro)] <texto>

Ejemplos

1. `#define PI 3.141592`
2. `#define cuadrado(x) x*x`
3. `#define area_circulo(r) PI*cuadrado(r)`

1. Se declara la constante `PI`. Cualquier aparición de `PI` en el programa será sustituida por el valor asociado: 3.141592.
2. Cuando `cuadrado(x)` aparece en una línea del programa, el preprocesador la sustituye por `x*x`.
3. La tercera directiva declara una operación matemática, una fórmula, que incluye las macros `cuadrado(x)` y `PI`. Calcula el área de un círculo.

Otros ejemplos

```
#define LONGMAX 81
#define SUMA(x,y)  (x)+(y)
```

El preprocesador sustituirá en el archivo fuente que contiene a estas dos macros por la constante `81` y la expresión `(x)+(y)` respectivamente. Así:

```
float vector[LONGMAX];          define un vector de 81 elementos.
printf("%d",SUMA(5,7));         escribe la suma de 5+7.
```

Nota: al ser una directiva del preprocesador no se pone al final ;(punto y coma)

A.21.2 Directiva `#error`

Esta directiva está asociado a la compilación condicional. En caso de que se cumpla una condición se escribe un mensaje.

Sintaxis **#error** texto
El `texto` se escribe como un mensaje de error por el preprocesador y termina la compilación.

A.21.2 Compilación condicional.

La compilación condicional permite que ciertas secciones de código sean compiladas dependiendo de las condiciones señaladas en el código. Por ejemplo, el código puede desearse que no se compile si se utiliza un modelo de memoria específico o si un valor no está definido.

Las directivas para la compilación condicional tienen la forma de sentencias if, son:
`#if`, `#elif`, `#ifndef`, `#ifdef`, `#endif`.

A.21.2.1 Directiva `#if, #elif, #endif`

Permite seleccionar código fuente para ser compilado. Se puede hacer selección simple o selección múltiple.

Formato 1

```
#if expresión_constante
      . . .
#endif
```

Se evalúa el valor de la *expresión_constante* .Si el resultado es distinto de cero, se procesa todas las líneas de programa hasta la directiva `#endif`; en caso contrario se saltan automáticamente y no se procesan por el compilador.

Formato 2

```
#if expresión_constante1
      . . .
#elif expresión_constante2
      . . .
#elif expresión_constanten
      . . .
#else
      . . .
```

```
#endif
```

Se evalúa el valor de la *expresión_constante*$_1$. Si el resultado es distinto de cero, se procesa todas las líneas de programa hasta la directiva `#elif`. En caso contrario se evalúa la *expresión_constante*$_2$, si es distinta de cero, se procesan todas la líneas hasta la siguiente `#elif` o `#endif`. En caso contrario se sigue evaluando la siguiente expresión, hasta que una sea distinta de cero, o bien, procesarse las líneas incluidas después de `#else`.

Ejemplo

```
#if VGA
      puts("Se está utilizando tarjeta VGA.");
#else
      puts("hardware de gráficos desconocido.");
#endif
```

Se escribe "Se está utilizando tarjeta VGA." si VGA es distinto de cero; en caso contrario se escribe la segunda frase.

A.21.2.2 Directiva `#ifdef`

Permite seleccionar código fuente según esté definida una macro (`#define`)

Formato `#ifdef identificador`

```
. . .
#endif
```

Si `identificador` está definido previamente con `#define identificador` entonces se procesan todas las líneas de programa hasta `#endif` ; en caso contrario no se compilan esas líneas. Al igual que la directiva `#if`, las directivas `#elif` y `#else` se pueden utilizar conjuntamente con `#ifdef` .

A.21.2.3 Directiva `#ifndef`

Ahora, la selección de código fuente se produce si no está definida una macro.

Formato `#ifndef identificador`

```
. . .
#endif
```

Si `identificador` no está definido previamente con `#define identificador` entonces se procesan todas las líneas de programa hasta `#endif` ; en caso contrario no se compilan esas líneas. Al igual que la directiva `#if`, las directivas `#elif` y `#else` se pueden utilizar conjuntamente con `#ifndef` .

Ejemplo

```
#ifndef _PAREJA
```

```
#define _PAREJA
      struct pareja {

            . . .
      }

      . . .

#endif
```

En el ejemplo si no está definida la macro `_PAREJA` se define y se procesa el código fuente que está escrito hasta `#endif`. Se evita que se procese el código más de una vez.

A.21.3 Directiva `#include`

Permite añadir código fuente escrito en un archivo al archivo que actualmente se está escribiendo.

Formato 1 `#include` *"nombre_archivo"*

El archivo *nombre_archivo* se incluye en el módulo fuente. El preprocesador busca en el directorio o directorios especificado en el path del archivo. Normalmente se busca en el mismo directorio que contiene al archivo fuente. Una vez que se encuentra se incluye el contenido del archivo en el programa, en el punto preciso que aparece la directiva `#include` .

Formato 2 `#include` <*nombre_archivo*>

El preprocesador busca el archivo especificado sólo en el directorio establecido para contener los archivos de inclusión.

A.21.4 Directiva `#line`

Formato `#line` *constante "nombre_archivo"*

La directiva produce que el compilador trate las líneas posteriores del programa como si el nombre del archivo fuente fuera *"nombre_archivo"* y como si el número de línea de todas las líneas posteriores comenzara por *constante* . Si *"nombre_archivo"* no se especifica, el nombre de archivo especificado por la última directiva `#line` , o el nombre del archivo fuente se utiliza (si ningún archivo se especificó previamente).

La directiva `#line` se utiliza principalmente para controlar el nombre del archivo y el número de línea que se visualiza siempre que se emite un mensaje de error por el compilador.

A.21.5 Directiva `#pragma`

Formato `#pragma` *nombre_directiva*

Con esta directiva se declaran directivas que usa el compilador de C. Si al compilarse el programa con otro

compilador de C, este no reconoce la directiva, se ignora .

A.21.6 Directiva `#undef`

Formato `#undef` *identificado*

El *identificador* especificado es el de una macro definida con `#define`, con la directiva `#undef` el *identificador* se convierte en no definido por el preprocesador. Las directivas posteriores `#ifdef`, `#ifndef` se comportarán como si el identificador nunca estuviera definido.

A.21.7 Directiva

Formato #

Es la directiva nula, el preprocesador ignora la línea que contiene la directiva #.

A.21.8 Identificadores predefinidos.

Los identificadores siguientes están definidos para el preprocesador:

Identificador	**Significado**
`_ _LINE_ _`	Número de línea actual que se compila.
`_ _FILE_ _`	Nombre del archivo actual que se compila
`_ _DATE_ _`	Fecha de inicio de la compilación del archivo ctual.
`_ _TIME_ _`	Hora de inicio de la compilación del archivo, en el formato "`hh:mm:ss`".
`_ _STDC_ _`	Constante definida como 1 si el compilador sigue el estándar ANSI C.