

I. APÉNDICE: Programación en C

I.1 Fundamentos

El lenguaje C tiene un determinado juego de instrucciones. Un programa en C está formado por un conjunto de funciones que reúnen las siguientes características:

- No se pueden anidar definiciones de función; es posible invocar desde una función a otra, pero no se puede anidar una función dentro de otra.
- Las funciones que componen un programa no tienen que estar necesariamente dentro de un mismo fichero fuente.
- Las funciones pueden ser recursivas.

El ejemplo siguiente muestra una pequeña función de nombre *main* que declara tres variables enteras y a continuación les asigna ciertos valores, y dado que no hay ningún identificador entre los paréntesis de apertura y cierre que acompañan al nombre de la función, se trata de una función que no recibe parámetros.

```
main() /* Esto es un comentario */
{
    int i, j, k; /* Declaración de tres variables enteras a las
                  que se asignará valores representados en
                  distintas bases de numeración */

    i = 10; /* Asigna a la variable "i" el valor decimal 10 */
    j = i + 015; /* Asigna a "j" el valor octal resultado de sumar
                  al valor de "i" 15 unidades octales */

    k = j * j + 0xFF; /* Asigna a k el resultado de sumar a "j"
                       multiplicado por "j", el valor hexadecimal FF
                       ("k" tendrá valor hexadecimal).

}
```

Todos los programas en C deben tener una función *main* que será la función principal, de manera que los programas comenzarán a ejecutarse a través de ésta. El cuerpo de las funciones en C se “encierra” entre una llave de apertura y una de cierre. Además, y tal y como puede observarse en el ejemplo anterior, en C pueden utilizarse variables,

siempre y cuando se declaren explícitamente antes de ser usarse. Así mismo, todas las instrucciones terminan en punto y coma (“;”) y los comentarios comienzan con los caracteres “/” y finalizan con “/” pudiendo extenderse varias líneas.

La función *main* del ejemplo declara tres variables de tipo entero (“i”, “j” y “k”). Posteriormente se les asignan unos valores, cada uno de ellos con una base de numeración diferente, de entre las tres más utilizadas en C:

- El valor 10 está una constante en base 10 (decimal);
- El valor o15 está expresado en base 8 (octal) y equivale a 13 en decimal (las expresiones numéricas octales comienzan siempre por “o”).
- La constante 0xFF está expresada en hexadecimal y equivale a 255 en decimal (las expresiones numéricas hexadecimales comienzan por “0x”).

1.2 Tipos de datos.

1.2.1. TIPOS BÁSICOS

En C, los tipos básicos de datos son los enteros, los caracteres y los decimales. El tipo “booleano” no existe, de manera que para simular su funcionamiento se utilizan los enteros, entendiendo todo valor igual a 0 como “falso” y todo valor distinto de 0 como verdadero.

Tipo	Palabra clave	Espacio necesario	Rango
Entero	int / short int	2 bytes	desde –32768 hasta 32767
	long	4 bytes	Desde –2147483648 hasta 2147483647
Carácter	char	1 byte	Desde – 128 hasta 127
Real	float	4 bytes	Desde 3,4 e –38 hasta 3,4 e 38
	double	8 bytes	Desde 1,7 e –380 hasta 1,7 e308
	long double	15 bytes	Desde –3,4 e-4932 hasta 1,1 e4932.

Tabla 14: Rangos por defecto de los distintos tipos básicos de datos.

Con estos tipos de datos pueden utilizarse los modificadores siguientes:

- **unsigned:** permite declara enteros o caracteres sin signo, de manera que el rango de un entero iría desde 0 hasta 65535.

- **signed:** permite declara enteros o caracteres con signo. De hecho, éste es el modificador que se utiliza por defecto aplicándose los rangos mostrados en la tabla.

Al declarar una variable también puede indicarse cómo debe almacenarse. Para ello se utilizan las palabras clave siguientes:

- **register:** se aplica a tipos enteros y caracteres. Indica al compilador que si puede, utilice un registro de la UCP en vez de una posición de memoria para almacenar la variable. De esta manera, se aumenta la velocidad del programa, aunque como los registros de UCP son muy limitados solamente debe emplearse con variables de acceso muy frecuente.
- **static:** indica que la variable se mantenga en memoria hasta que el programa acabe.
- **extern:** se utiliza en redeclaraciones de variables, para que el compilador no cree ningún espacio de almacenamiento adicional para ellas.

A continuación, se muestran algunos ejemplos de declaraciones de variables:

```
int i; /* variable de tipo entero */
short int z1, z2; /* dos enteros cortos */
char c; /* un carácter */
unsigned short int k; /* un entero corto sin signo */
long x; /* un entero largo; "int" puede omitirse */
register int r; /* una variable registro */
```

Conversiones

C tiene la capacidad de realizar conversiones, implícitas o explícitas (*cast*) entre distintos tipos de datos.

CONVERSIONES IMPLÍCITAS

Las conversiones implícitas se pueden realizar asignando un valor o una variable de un tipo de datos a una variable de otro tipo siempre y cuando la segunda tenga un rango de valores mayor que el original. Así con las siguientes declaraciones:

```
long entero_largo; /* Variable entero_largo de tipo entero largo */
int entero; /* Variable entero de tipo entero normal */
```

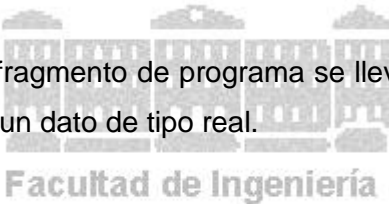
es posible realizar la asignación: `entero_largo = entero;` en el cuerpo de la función puesto que la variable *entero*, al tener un rango menor que la variable *entero_largo* también puede ser representada dentro del rango correspondiente a esta última; mediante 32 bits se puede representar una variable que sólo necesita 16 bits (si bien se desperdician los 16 bits de mayor peso).

La asignación inversa pasaría sin error por el compilador, pero provocaría una pérdida de información puesto que al pasar de 32 a 16 bits se truncaría una serie de bits. Por ello, se restringen las conversiones implícitas a rangos superiores al de partida.

CONVERSIÓN EXPLÍCITA

La mayoría de las conversiones se llevan a cabo automática o implícitamente si bien hay casos en los que es útil forzar la conversión de un tipo de datos a otro a través de lo que se denomina *casting* o conversión explícita. Para ello, se especifica entre paréntesis, antes del valor o variable a convertir, el tipo al cual va a ser transformada. Se utiliza especialmente en llamadas a funciones, dentro de expresiones o para dotar al código de claridad.

Por ejemplo, en el siguiente fragmento de programa se lleva a cabo la división de dos enteros para la obtención de un dato de tipo real.



```
void main()  
{  
    int dato1, dato2;  
    float cociente;  
    ...  
    cociente = (float) dato1 / (float) dato2;  
}
```

Este tipo de conversión es también de utilidad cuando se desea utilizar funciones ya declaradas:

Ejemplo 1:

```
long cuadrado (long i)  
{  
    return (i * i);  
}
```

En este caso, la función *cuadrado* recibe como argumento un valor de tipo *long int* y devuelve al punto desde el cual fue invocada un valor *long* que será el cuadrado del argumento recibido. Así, si se desea calcular el cuadrado del número entero 3 y almacenarlo en la variable *resultado_cuadrado* puede hacerse lo siguiente:

```
void main()
{
    long resultado_cuadrado;

    int numero;

    valor = 3;

    resultado_cuadrado = cuadrado ( (long) numero );
}
```

En esta función *main*, que no devuelve nada (su declaración va precedida de la palabra *void*) y no recibe ningún parámetro (entre los paréntesis tras su nombre no hay nada) se asigna a la variable *resultado_cuadrado* de tipo *long* el resultado devuelto por la función *cuadrado*. Además, con esta asignación se está haciendo una conversión de la variable *numero* a tipo *long* antes de ser pasado a la función *cuadrado*, ya que es el tipo de dato que espera.

Ejemplo 2:

```
int x;

int y;

x = 4 * (int) (5/2);

y = 4 * (5/2);
```

Si se comprueba el valor de estas variables tras esta secuencia se llegaría a la conclusión de que no son iguales. La variable “x” contiene el valor 8, mientras que la variable “y” contiene el valor 10. Esto se debe a la conversión realizada.

En conclusión, en ciertos puntos del código puede ser deseable forzar una conversión, mediante un *cast*. Además, también es útil para clarificar el código de programa, y así facilitar posteriores modificaciones y revisiones.

En las conversiones de tipos es conveniente prestar atención a las extensiones de signo. La conversión de enteros a caracteres puede depender de que el compilador considere los caracteres como números sin o con signo.

I.2.2. TIPOS COMPUESTOS

A partir de los tipos básicos de datos se pueden formar otros más complejos: vectores o arrays, registros o *structs*, registros de variables o datos de tipo *union* y punteros o *pointers*.

Arrays

Los vectores o arrays son conjuntos de elementos del mismo tipo. Existen vectores de varias dimensiones (1, 2, 3 ó más) pero en cualquier caso, el índice del array irá desde 0 hasta N-1, siendo N el valor indicado entre corchetes en la declaración del array. Así, si se declara por ejemplo `int a[10];` se está declarando un array de 10 enteros, que serán referidos desde el `a[0]` hasta el `a[9]`.

Es importante estaca que el nombre de un array sin corchetes dentro de un programa será interpretado como un puntero al primer elemento del array. Esta consideración es especialmente útil cuando se desea pasar un array como parámetro.

Registros (structs)

Los registros son conjuntos de variables, generalmente de tipos diferentes. La declaración siguiente:

```
struct
{
    int i;
    char c;
    { s;
```

declara “s” como un registro con dos miembros, el entero “i” y el carácter “c”. Para asignar por ejemplo el valor 6 al miembro “i” del registro se pondría `s.i = 6;` donde el operador punto indica que se ha seleccionado un miembro del registro.

Registros variantes (union)

También son registros formados también por varios miembros, pero a diferencia de los *struct* solamente pueden tomar el valor de uno de ellos en cada momento. Los compiladores reservarán pues espacio suficiente para albergar aquel miembro de estos tipos de datos que precise de mayor espacio en memoria. Así, la declaración

```
union
```

```
{
    int i;

    char c;

    { u;
```

indica que “u” puede contener un entero o un carácter, pero nunca ambos simultáneamente.

Punteros

En C, los punteros se utilizan para almacenar direcciones de memoria. Se utilizan muy frecuentemente y son declarados mediante el símbolo “*”. Así, la declaración `int i, *pi, a[10], *b[10], **pi;` declara un entero “i”, un puntero a un entero “pi”, un vector de 10 enteros “a”, un vector de 10 punteros a enteros “b” y un puntero a un puntero a un entero “pi”.

En la sentencia siguiente:

```
struct tabla
{
    int i;
    char *cp, c;
} z[20];
```



se declara un vector “z” de 20 registros, de maneja que cada uno de ellos tiene tres miembros: un entero “i”, un puntero a un carácter “cp” y un carácter “c”. Además en esta declaración se define el nombre *tabla* como de tipo registro, lo cual permite referirse al mismo en otras declaraciones utilizando la sintaxis *struct tabla*.

Así, `register struct tabla *p;` declara que “p” es un puntero a un dato de tipo *tabla*, indicando que debe ser depositado si es posible en un registro de la UCP. De esta manera, durante la ejecución del programa, “p” podría apuntar a cualquiera de los elementos de “z”, al ser éste de tipo *tabla*. Por ejemplo, escribiendo `p = &z[4]` “p” apuntaría a “z[4]”. El símbolo “&” es el operador unario de indirección, que podría ser sustituido por “obtén la dirección del elemento que me sigue”.

Posteriormente, para extraer uno de los elementos – por ejemplo el valor del miembro “i” – del registro apuntado por “p”, y almacenarlo después en la variable “n”, debería escribirse `n = p->i;` que podría interpretarse como “vete a la dirección apuntada por “p”, que contiene una estructura, y obtén el valor de su miembro “i”. Esta sentencia es

equivalente a `n = *p.i;` que podría interpretarse como “obtén el contenido apuntado por la dirección “*p*” y accede al miembro “*i*” del registro que contiene.

Lógicamente esta extracción también puede realizarse directamente de la variable “*z*”, escribiendo `n = z[4].i;`

Declaración de nuevos tipos de datos: typedef.

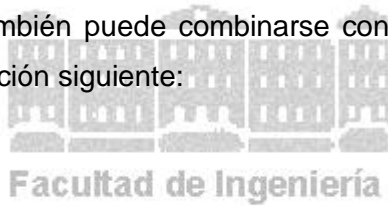
En ocasiones es conveniente dar nombres a los tipos compuestos. Para declarar un nuevo nombre para un tipo ya existente se utiliza la cláusula *typedef*. Así, `typedef unsigned short int unshort;` provoca que se defina el tipo de dato *unshort* que sería sinónimo del tipo *unsigned short int*. A partir de este momento es posible utilizar *unshort* como si de un tipo básico de dato se tratara. Por ejemplo, indicando:

```
unshort u1, *u2, u3[5]
```

se tendría un entero corto sin signo “*u1*”, un puntero a un entero corto sin signo “*u2*” y un vector de enteros cortos sin signo.

La palabra clave *typedef* también puede combinarse con *struct* para construir listas. Por ejemplo, dada la declaración siguiente:

```
struct nodo {
    char *nombre;
    char *apellidos;
    struct nodo *sig
}
```



con `struct nodo *primero;` se declararía un puntero a la lista resultante del *struct* anterior, pero si antes se declara `typedef struct nodo nodo;` se puede usar el nuevo tipo de datos – *nodo* – para definir los punteros necesarios con mayor facilidad. Por ejemplo, la declaración del puntero al primer nodo de la lista podría realizarse de la siguiente forma:

```
nodo *primero;
```

1.3 Instrucciones

En C, los procedimientos o funciones contienen tanto declaraciones como instrucciones. Una vez repasadas las declaraciones, se exponen las instrucciones *if*, *if...else*, *while* y *do...while*, *for*.

if (condición)

instrucción;

Ejemplo:

```
if (x < 0)
    K = 3;
```

if (condición)

```
{
    conjunto de instrucciones;
}
```

Ejemplo:

```
if (x > y)
{
    j = 2;
    k = j + 1;
}
```



if (condición)

parte if (si contiene más de una instrucción, irán encerradas entre llaves);

else

parte else (si contiene más de una instrucción, irán encerradas entre llaves);

Ejemplo:

```
if (x + 2 > y)
{
    j = 2;
    k = j - 1;
}
else
{
    m = 0;
```

```
k = j - 2;  
}
```

while (condición)

instrucción o conjunto de instrucciones entre llaves;

Ejemplo:

```
while (n > 0)  
    n = n - 1;
```

do

instrucción o conjunto de instrucciones entre llaves;

while (condición);

Ejemplo:

```
do {  
    k = k + k;  
    n = n - 1;  
} while (n > 0);
```



El funcionamiento de la instrucción *for* difiere del de otros lenguajes. Su forma genérica es la siguiente:

for (parte inicialización; condición; parte expresión)

instrucción o conjunto de instrucciones entre llaves;

pudiendo haber más de una sentencia tanto en *parte inicialización* como en *parte expresión*.

La sentencia *for* puede entenderse como:

parte inicialización;

while (condición)

```
{  
    instrucción o conjunto de instrucciones;  
    parte expresión;  
}
```

Un ejemplo podría ser:

```
for (i = 0; i < n; i = i + 1)

a[i] = 0;
```

Esta instrucción pone a 0 los “*n*” primeros elementos de un array “*a*”: comienza asignando a “*i*” el valor 0 (esta asignación se realiza una sola vez, como si no perteneciera al bucle); a continuación se itera, asignando a “*a[i]*” el valor 0 siempre y cuando al incrementar “*i*” en cada iteración su valor siga siendo menor que “*n*”.

Una instrucción similar al *case* de Pascal es la instrucción *switch* cuya sintaxis es la siguiente:

switch (expresión entera)

```
{
    case valor-entero 1: instrucción o conjunto de instrucciones 1;
                        break;
    case valor-entero 2: instrucción o conjunto de instrucciones 2;
                        break;
    ...
    default:            instrucción o conjunto de instrucciones default;
} /* fin del switch */
```

Se ejecutará cierta instrucción o conjunto de instrucciones en función del valor de la expresión entera que sigue a la cláusula *switch*. Si el valor de la expresión no coincide con ninguno de los especificados junto a cada sentencia *case* se selecciona para su ejecución la instrucción o conjunto de instrucciones escritas tras la cláusula *default*; si ésta no existe, simplemente se continúa con la instrucción inmediatamente posterior al *switch*.

Tras ejecutar las instrucciones correspondientes al *case* “activado” por la expresión entera, se continúa ejecutando las instrucciones del siguiente *case* de no encontrarse antes una sentencia *break*. Por ello, en la práctica, casi siempre hay que utilizar esta instrucción para que se ejecuten solamente las instrucciones correspondientes al *case* seleccionado.

La instrucción *break* también puede utilizarse dentro de los bucles *for* y *while*, haciendo que el programa salga del bucle. Si *break* se encuentra en el bucle más interno de una serie de bucles anidados, solamente abandona dicho nivel.

Una instrucción análoga es *continue*, pero no provoca la salida del bucle, sino que obliga a abandonar la iteración actual, para pasar a la siguiente. Puede interpretarse como un salto al principio del bucle.

1.4 Funciones

La función siguiente:

```
int suma (i, j)

int i, j;

{
    return (i + j);
}
```

devolverá un entero, como lo indica la palabra reservada *int* que precede al nombre de la función. Para la devolución de este valor debe emplearse la instrucción *return()*, que puede tener una expresión asociada, correspondiente al valor que devolverá la función al punto desde la cual fue invocada.

Existen restricciones para el paso de parámetros: no se permite que los parámetros de entrada a la función ni el dato (únicamente uno) a devolver pueda ser un array, un registro²¹ o un procedimiento, pero sí punteros a los mismos. De esta forma tanto los parámetros como el resultado quepan siempre en una única palabra de memoria. y las implementaciones serán más eficientes.

Por ejemplo, si “a” es el nombre de un array – sea del tipo que sea –, para pasar dicho array como parámetro a una función “g”, bastaría con pasar *g(a)* ;.

C no tiene instrucciones de entrada / salida incluidas en el propio lenguaje, de manera que las operaciones de E/S se llevan a cabo a través de procedimientos de biblioteca, siendo `printf ("x = %d y = %o z = %x\n", x, y, z);` el más común para las salidas.

El primer parámetro es una cadena de caracteres entre comillas dobles (verdaderamente es un array de caracteres). Cualquier carácter del citado array distinto del conjunto formado por el símbolo “%” y el carácter que le sigue, se imprime

²¹ En algunos compiladores, sí que es posible el paso de registros como parámetros.

tal cual. Al encontrar un %, se imprime el siguiente parámetro de la función *printf*, utilizando la letra que sigue al % para indicar la manera de imprimirlo:

%d ≡ imprimir como entero decimal.

%o ≡ imprimir como entero octal.

%u ≡ imprimir como entero decimal sin signo.

%x ≡ imprimir como entero hexadecimal.

%s ≡ imprimir como cadena de caracteres o string.

%c ≡ imprimir como carácter.

Es posible también la utilización de las letras D, O y X para imprimir valores de tipo *long* en decimal, octal o hexadecimal respectivamente.

Además, se pueden utilizar caracteres especiales que ayudan a generar secuencias de escape en pantalla, de los cuales los más utilizados son:

\n ≡ cambio de línea.

\r ≡ retorno de carro.

\t ≡ tabulador horizontal.

\v ≡ tabulador vertical.

\f ≡ salto de página.



1.5 Expresiones.

Las expresiones se forman combinando operandos y operadores. En C, existen operadores tanto aritméticos (+, -, ...) como de relación (<, >, ...) y su funcionamiento es semejante al de otros lenguajes.

Una particularidad de C es que permite combinar operadores y asignaciones, por ejemplo, se puede escribir `a + = 4;` que equivale a `a = a + 4;`.

El operador de igualdad es “==” diferenciándose así del de asignación (“=”) siendo el de desigualdad “!=”. De esta manera, para ver si dos variables son iguales se escribirá `if (a == b) expresión;`

Asimismo, existen operadores lógicos que permiten manipular los bits de una palabra, y por tanto realizar desplazamientos y operaciones booleanas bit a bit. Los operadores de desplazamiento son:

Izquierda: <<

Derecha: >>

Los operadores booleanos o lógicos bit a bit son:

Complemento: ~

AND: &

OR: |

XOR: ^

NOT: !

Este último operador devuelve 0 si su operando es distinto de 0 y 1 si dicho operando es igual a 0. Se suele utilizar fundamentalmente en instrucciones *if*, de la misma manera que en Pascal se emplea el operador *not*.

El operador ~ es el complemento bit a bit, es decir, calcula el complemento a 1 de su operando. Es decir, cada bit del operando a 0 se convierte en un 1 y viceversa.

El operador & extrae la dirección de una variable. Si, por ejemplo, "p" es un puntero a un entero e "i" es un entero, la instrucción `p = &i;` calcula la dirección en memoria de la variable "i" depositándola en la variable "p".

La operación contraria a la obtención de la dirección de un objeto, por ejemplo para dejarla en una variable puntero se logra con el operador "*". Así, por ejemplo, si se asigna la dirección de la variable "i" a la variable "p", `*p` tiene el mismo valor que "i".

El operador *sizeof* calcula el tamaño en bytes de su operando. Por ejemplo, si se aplica a un array *a*, de 20 enteros, en una máquina con enteros de 2 bytes, *sizeof(a)* devolvería el valor 40.

El operador "?" selecciona entre dos alternativas separadas por el símbolo ":". Por ejemplo, `i = (x < y ? 6 : k + 1);` compara las variables "x" e "y", de manera que si la primera es menor que la segunda asigna a "i" el valor 6, y en caso contrario, el valor "`k + 1`".

Otros dos operadores de C son el de incremento y decremento. Así, `p++;` indica que se incremente a "p" en una cantidad que depende de su tipo. Los enteros y los caracteres se incrementan en una unidad, mientras que los punteros se incrementan en una cantidad igual al tamaño del objeto al que apuntan; es decir, si "a" es un array de registros y "p" es un puntero a uno de ellos, `p = &a[3];` convierte a "p" en un

apuntador al cuarto elemento del vector. Tras incrementar “*p*”, éste apuntaría a “*a*[4]”. El operador de decremento es análogo, expresado como “- -”.

Estos operadores se pueden combinar con una asignación pues según se escriben antes o después de su operando, el significado será de pre o post incremento / decremento respectivamente. En el caso de $n = k++$; siendo ambas variables enteras, en primer lugar se asigna a “*n*” el valor de “*k*” y posteriormente se incrementa la variable “*k*” en una unidad. En cambio, si se escribe $n = ++k$; primero se incrementa el valor de “*k*” en una unidad y posteriormente se almacena en “*n*” su nuevo valor.

1.6 El preprocesador de C

Los programas fuentes en lenguaje C, antes de pasarse al compilador, pasan automáticamente por otro programa denominado *preprocesador*. La salida generada por este preprocesador constituye la entrada del compilador, en vez del programa fuente original. Este preprocesador realiza tres transformaciones importantes sobre el programa fuente:

Inclusión de ficheros.

Definición y expansión de macros.

Compilación condicional.

1.6.1. INCLUSIÓN DE FICHEROS.

Dentro de los programas se puede especificar que determinadas líneas van dirigidas al preprocesador poniendo como primer carácter el símbolo “#” y no incluyendo un “;” al final. Así, si ante el comando:

```
# include "fichero.h"
```

el preprocesador incluye, en el programa que luego pasará al compilador el código del fichero especificado, línea a línea.

Para ello, se busca el fichero especificado – en este caso “fichero.h” – en el directorio de trabajo (así lo determinan las comillas dobles). Sin embargo, si el comando es:

```
# include <fichero.h>
```

el fichero a incluir se buscará en el directorio */usr/include*.

En C, es una práctica común agrupar las declaraciones utilizadas por varios ficheros de código en ficheros de cabecera (cuya extensión suele ser “.h”) que posteriormente son *incluidos* en los ficheros de código mediante el comando *include*.

I.6.2. DEFINICIÓN Y EXPANSIÓN DE MACROS.

El preprocesador permite también la definición de macros. Por ejemplo:

```
# define TAMANO_BLOQUE 1024
```

define la macro TAMANO_BLOQUE asignándole el valor 1024. A partir de ese momento, cualquier aparición en el fichero fuente de la cadena de caracteres “TAMANO_BLOQUE” será reemplazada – antes de que el fichero sea pasado al compilador – por la cadena de caracteres “1024”, es decir, se sustituye una cadena de caracteres por otra.

Por convenio, los nombres de las macros se escriben siempre en mayúsculas, e incluso existe la posibilidad de que tengan parámetros, si bien esta posibilidad raramente se utiliza.

I.6.3. COMPILACIÓN CONDICIONAL

La tercera función del preprocesador es la compilación condicional. Existen casos en que el código destinado a un preprocesador determinado difiere del destinado al resto, de manera que en función del procesador habrá códigos que deberán o no ser compilados. Estos fragmentos de código presentan el siguiente aspecto:

```
# ifdef i8088

instrucciones a compilar solamente en el Intel 8088

# endif
```

Si el símbolo *i8088* está definido, el preprocesador hará que estas instrucciones sean añadidas a su salida – recordemos que esta salida constituirá la entrada al compilador – en caso contrario, estas instrucciones serán omitidas.