Universidad de Buenos Aires
Facultad De Ingeniería

Programando en C a Bajo Nivel

[75.40] Algoritmos y Programación I

1er Cuatrimestre 2011

Cátedra: Ing. Pablo Guarna

Autor: Bernardo Ortega Moncada

Versión 3.0

Índice

1. Introducción

Este apunte esta orientado para que el lector pueda apreciar como uno puede trabajar en el lenguaje de programación C con sentencias de Bajo Nivel y utilizar herramientas que pueden ser útiles a la hora de programar sentencias que podrían simplificar lineas operando a bajo nivel. Este apunte es de carácter informativo, no significa que lo vayan a usar en esta materia, pero quizás en alguna posterior. Esta demas aclarar que para leer este apunte se necesita un conocimiento básico previo del Lenguaje. Todas estas operaciones que se muestran en el apunte, también son válidas en el **Lenguaje C++**

2. Representación de Números en Base Binaria

Como uno ya sabe, los números en la computadora no se expresan de la misma forma que un ser humano escribe. Ya que la computadora trabaja de forma binaria y solo puede comprender si algo es verdadero o falso, si hay información o no, por lo tanto si uno esta trabajando con números que ocupan 1 Byte, uno sabe que 1 Byte esta compuesto por 8 bits, lo cual un bit solo puede almacenar 1 o 0.

Por lo tanto si nosotros a modo de ejemplo tenemos el número, la computadora ve a este número de la siguiente manera:

Representación Decimal		Representación Binaria en 8 bits
$0\|_{10}$	=	$00000000\|_2$
$2\|_{10}$	=	$00000010\|_2$
$7\|_{10}$	=	$00000111\|_2$
$25\|_{10}$	=	$00011001\|_2$
$255\|_{10}$	=	$11111111\|_2$

Pero tenemos que saber que un número representado en binario posee 2 bits muy importantes. **El bit mas Significativo (MSB, por sus siglas en ingles)** y **El Bit Menos Significativo (LSB, por sus siglas en ingles)**.

Una observación a tener en cuenta es que el **MSB** también es considerado Bit de Signo, siempre y cuando se use la representación **MyBS** (Módulo y Bit de Signo).

- se considera al 1 como el signo negativo
- se considera al 0 como el signo positivo

Por ejemplo, tenemos un numero representado en su forma binaria, podemos apreciar cuales son los bits significativos:

$$\underset{\textbf{MSB}}{\boxed{1}}\ 0\ 0\ 1\ 0\ 1\ 0\ \underset{\textbf{LSB}}{\boxed{1}}$$

2.1. Complementos A1 de un Número

El complemento A1 de un numero representado en forma binaria, consiste en negar (o invertir) bit a bit la expresión del número, en lineas generales consiste en cambiar los 0's por 1's y viceversa. Esta herramienta es muy poderosa en el ámbito de la computación ya que nos permite representar un número de forma negativa, ya que como sabemos, la computadora no reconoce los números negativos. Veamos el siguiente ejemplo de como calcular el complemento A1 a la base de un numero binario cualquiera:

$$01100100 \Rightarrow \underbrace{10011011}_{\text{Complemento A1}}$$

3. Sentencias Bit a Bit en C

Uno puede hacer una serie de operaciones lógicas bit a bit con el lenguaje C, e incluso realizar complementos A1, o desplazamientos de Bits a la derecha o a la izquierda. Estas operaciones son muy comunes en el lenguaje de bajo nivel denominado Assembler (Ensamblador). A continuación se muestra una tabla con algunas de las operaciones a bajo nivel que podemos realizar:

Operador	Acción
&	Operación AND (bit a bit) simula una compuerta AND - 74ls08 TTL
|	Operación OR (bit a bit) simula una compuerta OR - 74ls32 TTL
∧	Operación XOR (bit a bit) simula una compuerta XOR - 74ls86 TTL
∼	Complemento A1 a la base
>>	Desplazamiento de bits hacia la derecha
<<	Desplazamiento de bits hacia la izquierda
? :	Operador Ternario

3.1. Operación AND (Bit a Bit) (&)

Si recordamos que la operación AND (Bit a Bit) esta definida por la siguiente tabla de verdad:

A	B	$A\&B$
0	0	0
0	1	0
1	0	0
1	1	1

Como trabajamos con variables que ocupan 1 Byte (8 bits) la operación AND Bit a Bit para estos casos se comportaría de la siguiente manera:

Supongamos que tenemos una variable $a = 23$ y otra variable $b = 90$, su representación en binario de ambas seria:

$a = 00010111$ y $b = 01011010$
por lo tanto:

$$\begin{array}{r} 00010111 \\ \& \\ 01011010 \\ \hline 00010010 \end{array}$$

El resultado nos da un número en binario cuya expresión es $00010010|_2$, que en decimal es conocido como $18|_{10}$.

Por lo tanto llevando este ejemplo al **lenguaje C**, se realizaría de la siguiente manera:

Ejemplo1: Operación Lógica AND Bit a Bit

```
1  #include <stdio.h>
2
3  int main()
4  {
5          char a = 23;
6          char b = 90;
7          char resultado;
8
9          printf(''a = %d  b = %d'',a,b);
10
11         resultado = a & b;
12
13         printf(''Resultado = %d'',resultado);
14
15         return 0;
16 }
```

Una aplicación útil para este operador, es si queremos averiguar si un bit de cierto número es 1 o 0, por ejemplo, se tiene el número $a = 75$ y se quiere averiguar si el cuarto bit de dicho número es 1 o 0. Para eso tenemos que aplicar el operador & al número a con un número cuya representación binaria sea 00001000, dicho número es el 8.

Veamos como queda la operación primero:

$$\begin{array}{r} 01001011 \\ \& \\ 00001000 \\ \hline 00001000 \end{array}$$

Entonces el código nos quedaría de la siguiente forma:

Ejemplo2: Operación Lógica AND Bit a Bit

```c
#include <stdio.h>

int main()
{
        char a = 75;
        char b = 8;
        char resultado;

        printf(''a = %d  b = %d'',a,b);

        if ( a & b)
                printf(''el cuarto bit de la variable a es 1 \ n'');
        else
                printf(''el cuarto bit de la variable a es 0 \ n'');

        return 0;
}
```

3.2. Operación OR (Bit a Bit) (|)

Si recordamos que la operación OR (Bit a Bit) esta definida por la siguiente tabla de verdad:

A	B	$A\|B$
0	0	0
0	1	1
1	0	1
1	1	1

Como trabajamos con variables que ocupan 1 Byte (8 bits) la operación AND Bit a Bit para estos casos se comportaría de la siguiente manera:

Supongamos que tenemos una variable $a = 23$ y otra variable $b = 90$, su representación en binario de ambas seria:

$a = 00010111$ y $b = 01011010$
por lo tanto:

$$\begin{array}{r} 00010111 \\ | \\ 01011010 \\ \hline 01011111 \end{array}$$

El resultado nos da un número en binario cuya expresión es $01011111|_2$, que en decimal es conocido como $95|_{10}$.

Ejemplo: Operación Lógica OR Bit a Bit

```c
#include <stdio.h>

int main()
{
        char a = 23;
        char b = 90;
        char resultado;

        printf(''a = %d  b = %d'',a,b);

        resultado = a | b;

        printf(''Resultado = %d'',resultado);

        return 0;
}
```

3.3. Operación XOR (bit a bit) ($\wedge$)

Si recordamos que la operación XOR (Bit a Bit) esta definida por la siguiente tabla de verdad:

A	B	$A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

Como trabajamos con variables que ocupan 1 Byte (8 bits) la operación XOR Bit a Bit para estos casos se comportaría de la siguiente manera:

Supongamos que tenemos una variable $a = 23$ y otra variable $b = 90$, su representación en binario de ambas seria:

$a = 00010111$ y $b = 01011010$
por lo tanto:

$$\begin{array}{r} 00010111 \\ \wedge \\ 01011010 \\ \hline 01001101 \end{array}$$

El resultado nos da un número en binario cuya expresión es $01001101|_2$, que en decimal es conocido como $77|_{10}$.

Ejemplo: Operación Lógica XOR Bit a Bit

```
1  #include <stdio.h>
2
3  int main()
4  {
5          char a = 23;
6          char b = 90;
7          char resultado;
8
9          printf(''a = %d  b = %d'',a,b);
10
11         resultado = a ^ b;
12
13         printf(''Resultado = %d'',resultado);
14
15         return 0;
16 }
```

3.4. Operación Complemento A1 a la Base ($\sim$)

Supongamos que tenemos una variable $a = 23$ su representación binaria seria:

$a = 00010111$
por lo tanto:

$$00010111 \quad \sim \quad 11101000$$

El resultado nos da un número en binario cuya expresión es $11101000|_2$, que en decimal es conocido como $-24|_{10}$. Hay que tener en cuenta que el **MSB** se considera como bit de signo

Ejemplo: Complemento A1 a la base

```
1  #include <stdio.h>
2
3  int main()
4  {
5          char a = 23;
6          char resultado;
7
8          printf(''a = %d'',a);
9
```

```
10          resultado = ~a;
11
12          printf(''Resultado = %d'',resultado);
13
14          return 0;
15  }
```

3.5. Desplazamiento de Bits

Este tema no es facil de entender al principio, pero para que quede claro, uno puede hacer corrimientos de N Bits, con $N \in \mathbb{N}$, es decir $N = 1, 2, \ldots$. Esta operación sirve básicamente para modificar un número a nivel Bit. Los corrimientos de Bits, pueden ser en dos direcciones:

- Corrimiento hacia la derecha
- Corrimiento hacia la izquierda

3.5.1. Desplazamiento de Bits a la Derecha (>>)

Supongamos que tenemos un Byte llamado $a = 01101001 \Rightarrow 105|_{10}$, a este número se le puede ser corrimientos de bits de la siguiente manera:

Número	Cantidad de Bits de Corrimiento	Resultado
01101001	1	$00110100 \Rightarrow 52\|_{10}$
01101001	2	$00011010 \Rightarrow 26\|_{10}$
01101001	3	$00001101 \Rightarrow 13\|_{10}$
01101001	4	$00000110 \Rightarrow 6\|_{10}$
01101001	5	$00000011 \Rightarrow 3\|_{10}$

Por lo tanto la función para aplicarlo en el **Lenguaje C**, es:

numero >> cantidad bits de corrimiento;

Ejemplo: Desplazamiento a la Derecha de Bits

```
1  #include <stdio.h>
2
3  int main()
4  {
5          char a = 105;
6          char resultado;
7
8          printf(''a = %d'',a);
9
10         resultado = a >> 1;
11
12         printf(''Resultado = %d'',resultado) /*imprime 52*/;
13
14         resultado = a >> 2;
15
16         printf(''Resultado = %d'',resultado) /*imprime 26*/;
17         return 0;
18 }
```

En pocas palabras, los corrimientos de Bits hacia la derecha, consisten en realizar la división Entera del número al cual se le realiza dicho corrimiento, es decir:

numero >> cantidad bits de corrimiento; $\approx$ numero % (cantidad bits de corrimiento + 1);

3.5.2. Desplazamiento de Bits a la Izquierda (<<)

Supongamos que tenemos un Byte llamado $a = 01101001 \Rightarrow 105|_{10}$, a este número se le puede ser corrimientos de bits de la siguiente manera:

01101001	1	$11010010 \Rightarrow 210\|_{10}$
01101001	2	$10100100 \Rightarrow 420\|_{10}$
01101001	3	$01001000 \Rightarrow 840\|_{10}$
01101001	4	$10010000 \Rightarrow 1680\|_{10}$
01101001	5	$00100000 \Rightarrow 3360\|_{10}$

Nota: *Pueden apreciar que los valores obtenidos en la base decimal no coinciden en lo absoluto con su representación binaria, esto se debe al* ***overflow*** *que existe al realizar un desplazamiento de bits, lo cual para realizar esto, hay que tener sumo cuidado!!*

Por lo tanto la función para aplicarlo en el **Lenguaje C**, es:

numero << cantidad bits de corrimiento;

Ejemplo: Desplazamiento a la Izquierda de Bits

```
1
2  #include <stdio.h>
3
4  int main()
5  {
6          char a = 105;
7          char resultado;
8
9          printf(''a = %d'',a);
10
11         resultado = a << 1;
12
13         printf(''Resultado = %d'',resultado) /*imprime 210*/;
14
15         resultado = a << 2;
16
17         printf(''Resultado = %d'',resultado) /*imprime 420*/;
18         return 0;
19 }
```

En pocas palabras, los corrimientos de Bits hacia la izquierda, consisten en realizar **la multiplicación** del número al cual se le realiza dicho corrimiento, es decir:

numero << cantidad bits de corrimiento; $\approx$ numero $*$ (cantidad bits de corrimiento + 1);

3.6. Operador Ternario (? :)

Este operador también resulta complejo de comprender al principio, pero para entenderlo, tenemos que verlo desde el punto de vista de un **IF/ELSE**. Ya que básicamente es la **expresión abstracta** de dicha operación. El operador ternario funciona con tres expresiones. Supongamos que tenemos tres expresiones (que claramente tienen que ser Expresiones que arrojen resultados "booleandos" -hay que recordar que en C no existen los booleanos-) $E1, E2$ y $E3$. Si queremos realizar operaciones dependiendo del estado de $E1$, uno diría, bueno acudamos a nuestro querido IF, entonces decimos que si $E1$ = TRUE, entonces se ejecute la expresión $E2$, sino que se ejecute la expresión $E3$. Por lo tanto el código nos quedaría de la siguiente manera:

Utilizando un IF

```
1  /*codigo anterior*/
2  if (E1)
3          E2
4  else
5          E3
6  /*codigo posterior*/
```

Ahora veamos como queda esto con el **Operador Ternario**

Utilizando un Operador Ternario

```
1 /*codigo anterior*/
2 E1 ? E2 : E3;
3 /*codigo posterior*/
```

Entonces podemos analizar lo siguiente:

- E1 ? → representa básicamente if(E1)
- E2 → como vemos que E2 esta despues de "?", entonces seria la expresión que se ejecutaría el caso en que E1 = TRUE
- : E3 → Como vemos que E3 esta después de ":", entonces tomemos como el signo ":" la representación abstracta de un **ELSE**, que luego la expresión posterior sería la que se ejecutaría en caso de que E1 = FALSE.

3.6.1. Ejemplo

Veamos un ejemplo del Operador Ternario. Supongamos que queremos hacer un algoritmo que nos diga si un número es Par o Impar, para eso uno haría el clásico algoritmo que todo el mundo conoce. El de aplicar división entera por 2 y verificar si el resto da cero o no. Ahora como nosotros sabemos manejar el lenguaje a un bajo nivel, vamos a realizar esta operación a nivel Bit. Es decir, vamos a verificar que el **LSB** de un número sea 1 o 0; Porque realizo esta alocada manera de calcular un número par o impar? Básicamente porque si un número cuya representación binaria tiene que su **LSB** = 1, entonces el número es impar, sino, si su **LSB** = 0, entonces el número es par.

Por lo tanto el ejemplo nos quedaría de la siguiente manera:

Algoritmo de un Número Par a bajo nivel

```
1 #include <stdio.h>
2
3 int main()
4 {
5         char a;
6         printf("ingrese un numero \n");
7         scanf("%d",&a);
8         (a & 1) ? printf ("el numero es impar \n") : printf("el numero es par \n") ;
9         return 0;
10 }
```

3.7. Campos de Bits

El método que se utiliza en C para operar con campos de bits, está basado en las estructuras (structs) que todos conocemos del lenguaje. Una estructura compuesta por bits se denomina campo de bits. La forma general de definición de un campo de bits es:

Estructura genérica de campos de bits

```
1 typedef struct s_bits
2 {
3         <tipo_de_dato> nombre1: <cantidad_de_bits>;
4         <tipo_de_dato> nombre2: <cantidad_de_bits>;
5         ...
6         <tipo_de_dato> nombre3: <cantidad_de_bits>;
7 }campo_bits;
```

Cada campo de bits puede declararse como char, unsigned char, int, unsigned int, etc y su longitud puede variar entre 1 y MAX NUMEROS DE BITS ACORDE A LA ARQUITECTURA DEL MICROPROCESADOR EN QUE SE ESTE TRABAJANDO DE LA VARIABLE. Es decir, supongamos que trabajamos en una arquitectura de 32 bits, sabemos que un char o unsigned char ocupa 1 Byte (8 bits), por lo tanto para el char, el campo de bits puede variar entre 1 bit y 8 Bits. Ahora, sabemos que un int o unsigned int, ocupa (en esta arquitectura) 32 bits, por lo tanto, este campo de bits puede variar entre 1 bit y 32 bits.

Veamos el siguiente ejemplo de campos de bits, que consiste en

Ejemplo1: Construcción de Campos de Bits

```
1 struct s_bits
2 {
3         unsigned int mostrar: 1;
4         unsigned int rojo: 2;
5         unsigned int azul: 2;
```

```
6         unsigned int verde: 2;
7         unsigned int transparencia: 1;
8 }campo_bits;
```

Esta estructura define cinco variables. Dos de ellas ("mostrar" y "transparencia") son de 1 bit, mientras que el resto son de 2 bits. Se puede utilizar esta estructura u otra similar para codificar una imagen **RGB** de 16 colores, especificando para cada píxel su color, si es o no transparente y si debe o no mostrarse. Para acceder a cada variable del campo de bits se emplea la misma sintaxis que en cualquier otra estructura de C común y corriente. Pero hay que tener en cuenta que un bit sólo puede tomar dos valores, 0 y 1 (en el caso de los campos de dos bits, serían cuatro los valores posibles: 0, 1, 2 y 3). Ya que si no tenemos esto en cuenta, podemos llegar a tener errores en tiempo de compilación.

Ejemplo2: Uso de Campos de Bits

```
1 #include <stdio.h>
2
3 struct s_bits
4 {
5         unsigned int mostrar: 1;
6         unsigned int rojo: 2;
7         unsigned int azul: 2;
8         unsigned int verde: 2;
9         unsigned int transparencia: 1;
10 }campo_bits;
11
12 int main()
13 {
14         campo_bits unColor;
15         /*creo un color aleatorio*/
16         unColor.rojo = 2;
17         unColor.azul = 1;
18         unColor.verde = 3;
19         unColor.transparencia = 0;
20
21         /*si el color no es transparente, entonces activo el campo para mostrarlo en alguna operacio
22
23         if (unColor.transparencia == 0)
24         {
25                 unColor.mostrar = 1;
26         }
27
28         /*aqui vienen otras sentencias*/
29
30         return 0;
31 }
```

3.7.1. Limitaciones de Campos de Bits

Las variables de campos de bits tienen ciertas restricciones, veamos algunas de ellas:

- No se puede aplicar el operador & sobre una de estas variables para averiguar su dirección de memoria (es decir, de algún elemento del struct).
- No se pueden construir arrays de un campo de bits (es decir, de algún elemento del struct).
- En algunos entornos, los bits se dispondrán de izquierda a derecha y, en otras, de derecha a izquierda, por lo que la portabilidad puede verse comprometida!!!.

3.8. Ventajas y Desventajas de Operar Bit a Bit

Las ventajas que podemos obtener son:

- Mayor velocidad de procesamiento al momento de ejecutar el programa.
- Mas ahorro de memoria en caso de querer modificar variables.
- Son ventajosos al momento de querer programar Microcontroladores (o Microprocesadores).
- Ocupan poco espacio.

Las desventajas que podemos obtener son:

- El código se puede tornar menos legible y bastante confuso.
- Puede quitarle portabilidad al programa lo cual, no es muy agradable.
- Seguido de la anterior, son dependientes de la arquitectura en donde se este trabajando.
- Uno es mas libre a cometer errores y lograr facilmente overflow en la memoria de cualquier dispositivo en el que se esté programando.

4. El Preprocesador de C

El preprocesador de C, es el primer programa invocado por el compilador y procesa directivas como:

- # include
- # define
- # ifndef
- # if
- etc.

El preprocesador utiliza 4 etapas denominadas Fases de traducción. Aunque alguna implementación puede elegir hacer algunas o todas las fases simultaneamente, debe comportarse como si fuesen ejecutadas paso a paso.

1. **Tokenizado léxico**: El preprocesador reemplaza la secuencia de trigrafos por los caracteres que representan (en pocas lineas, consisten en reemplazar símbolos en los cuales no son aplicados en algunos países, como por ejemplo $\sim$ o el identificador de un macro, etc).
2. **Empalmado de líneas**: Las líneas de código que continúan con secuencias de escape de nueva línea son unidas para formar líneas lógicas (ejemplo reconoce cuando puede o no haber un fin de linea $\backslash n$).
3. **Tokenización**: Reemplaza los comentarios por espacios en blanco. Divide cada uno de los elementos a preprocesar por un carácter de separación.
4. **Expansión de macros y gestión de directivas**: Ejecuta las líneas con directivas de preprocesado incluyendo las que incluye otros archivos y las de compilación condicional. Además expande las macros.

Pero en definitiva, el preprocesador es un programa separado que es invocado por el compilador antes de que comience la traducción real. Un preprocesador de este tipo puede eliminar los comentarios, incluir otros archivos y ejecutar sustituciones de macros.

Ahora bien, la pregunta del millón que quizás surja es la siguiente **¿Que es un Macro?**. Para eso vamos a dar la definición de Macro.

4.1. ¿Que Es Un Macro?

Un Macro, es una herramienta que nos permite evitar al programador la tediosa repetición de partes idénticas de un programa, los ensambladores y compiladores cuentan con macroprocesadores que permiten definir una abreviatura para representar una parte de un programa y utilizar esa abreviatura cuantas veces sea necesario. Para utilizar una macro, primero hay que declararla. En la declaración se establece el nombre que se le dará a la macro y el conjunto de instrucciones que representará.

El programador escribirá el nombre de la macro en cada uno de los lugares donde se requiera la aplicación de las instrucciones por ella representadas. La declaración se realiza una sola vez, pero la utilización o invocación del macro puede hacerse cuantas veces sea necesario. La utilización de macros posibilita la reducción del tamaño del código fuente, aunque el código objeto tiende a ser mayor que cuando se utilizan funciones.

4.2. Ejemplos de Macros

4.2.1. Inclusión de Archivos

Este caso es el mas común y el que se utiliza con mayor frecuencia (por no decir **siempre**). Uno siempre incluye archivos cuya extención es ".h" ya que son considerados **"Archivos de Cabecera"** o **"Headers Files"**, el mas famoso de todos es el archivo **"stdio.h"** y el operador que nos permite incluir este tipo de archivos es:

- #include

lo cual para utilizarlo en el **Lenguaje C**, tenemos que realizar:

Ejemplo: Inclusión de Archivos por el Preprosesador

```
1 #include <stdio.h> /*para los archivos de cabecera predefinidos en C*/
2 #include ''miarchivocabecera.h'' /*para los archivos de cabecera creados por uno mismo*/
3
4 int main()
5 {
6         printf(''Hola Mundo \n'');
7         return 0;
8 }
```

4.2.2. Creación de Archivos de Cabecera

Uno puede crear sus propios archivos de cabecera para poder simplificar el código fuente en el archivo **"main.c"**, básicamente es el caso análogo cuando uno creaba **UNITS en PASCAL** Para eso, utilizamos las siguientes sentencias:

- #ifndef (verifica si el macro no fue creado)
- #ifdef (verifica si el macro fue creado)
- #define (crea el macro)
- #include (si es necesario incluir otros archivos de cabecera)
- #endif (marca el fin de un #ifndef o #ifdef)

lo cual para utilizarlo en el **Lenguaje C**, tenemos que crear un archivo ".h" :

Ejemplo: Creación de un header file propio

```
1  /*si no esta definido el macro __MIARCHIVOCABECERA_H__, lo crea y luego llama el archivo stdio.h*/
2  #ifndef __MIARCHIVOCABECERA_H__
3  #define __MIARCHIVOCABECERA_H__
4
5  #include <stdio.h> /*incluyo un archivo*/
6
7
8  /*si ya esta definido el macro __MIARCHIVOCABECERA_H__, incluye mas archivos*/
9  #ifdef __MIARCHIVOCABECERA_H__
10 #include <stdio.h>
11 #include <string.h>
12 #include <malloc.h>
13 #endif /*fin del #ifdef*/
14
15 /*aca puedo definir tipos de datos*/
16
17 /*aca puedo definir las firmas de funciones/procedimientos*/
18
19 #endif /*fin del #ifndef*/
```

4.2.3. Creación de Macros como Constantes

Estos tipos de Macros son conocidos como **Macro Objetos**. Para poder definir un Macro Objeto en C, se tiene que tener presente como está compuesto. Para eso, veamos la estructura "genérica" de un Macro Objeto:

Estructura de un Macro Objeto

```
#define <nombre del macro> <lista de tokens a reemplazar en el codigo>
```

Veamos un ejemplo de como se construye un Macro Objeto:

Ejemplo: Construcción de un Macro Objeto

```
#include <stdio.h>
#define PI  3.14159
#define SALUDO ''Hola Mundo''

int main()
{
        printf(''%f'', PI); /* Esto imprime 3.14159 por pantalla */
        printf(SALUDO); /*Esto imprime Hola Mundo por pantalla*/

        return 0;
}
```

Podemos apreciar que:

- <nombre del macro> es PI
- <lista de tokens a reemplazar en el código> es 3.14159

En definitiva creamos un Macro Objeto llamado PI, que en cualquier parte del código donde aparezca "PI", será reemplazado por 3.14159. Esta ventaja de crear este tipo de Constantes, es que no ocupan lugar en memoria, sino que son reemplazados en tiempo de compilación, haciendo que el programa sea un poco mas liviano!!

4.2.4. Creación de Macros como Función

Estos tipos de Macros son conocidos como **Macro Funciones** (valga la redundancia). Para poder definir una Macro Función en C, se tiene que tener presente como está compuesto. La estructura genérica de una Macro Función no difiere casi en nada a la de un **Macro Objeto**, sólo tiene una diferencia y es que toda sentencia dentro de una Macro Función tiene que ir entre paréntesis ( ) aunque una gran ventaja es que pueden hacerse macros multilinea, es decir, que pueden hacer macros con mas de una linea de código, siempre y cuando antepongan el símbolo "\" al final de cada linea. Veamos un ejemplo de como crear una Macro Función, como por ejemplo, una Macro Función que calcule si un número es mas grande que otro.

Ejemplo: Construcción de una Macro Función

```
#include <stdio.h>

#define MAX(a,b) ( (a > b)? a : b )

int main()
{
        int num1;
        int num2;

        printf(''ingrese un numero \n'');
        scanf(''%d'',&num1);
        printf(''ingrese otro numero \n'');
        scanf(''%d'',&num2);

        printf(''El maximo entre %d y %d es: %d \n '',num1,num2,MAX(num1,num2));

        return 0;
}
```

El único problema que presenta realizar una Macro Función de esta manera, es que evalúa 2 veces a cualquiera de las 2 variables (en este caso a o b), dependiendo la condición que cumpla, lo cual no es del todo prolijo y hay que tener sumo cuidado cuando uno quiere hacer una Macro Función que modifique una variable, ya que podría modificarla dos veces y no sería un resultado del todo agradable. Si se esta trabajando con **ANSI C**, este problema es inevitable, lo cual no hay manera de salvar este problema. Por lo tanto si se quiere realizar Macro Funciones en **ANSI C**, hay que tener mucho cuidado y pensarlo dos veces si esa función conviene o no hacerla en Macro.

Si se trabaja en otra estandarización de C, como por ejemplo **C99**, uno puede salvar este problema con una sentencia llamada "**tipeof()**".

Dicha sentencia lo que hace es averiguar el tipo de dato que compone la variable que se le introduzca, y así evita que haya una sobrevaluación de dicha variable (igual a esto hay que tomarlo con pinzas, ya que le estaríamos quitando portabilidad al programa).

Entonces el ejemplo anterior quedaría como:

Solución al problema

```
#include <stdio.h>
#define MAX(a,b) \
                ({ typeof (a) _a = (a); \
                typeof (b) _b = (b); \
                _a > _b ? _a : _b; })

int main()
{
        int num1;
        int num2;

        printf(''ingrese un numero \n'');
        scanf(''%d'',&num1);
        printf(''ingrese otro numero \n'');
        scanf(''%d'',&num2);

        printf(''El maximo entre %d y %d es: %d \n '',num1,num2,MAX(num1,num2));

        return 0;
}
```

Veamos también como podemos realizar la función que calcula si un número es Par o No a bajo nivel por medio de Macro Funciones:

Macro Función Bajo Nivel para averiguar si un número es Par o No

```
#include <stdio.h>
#define PAR(a) \
        ( (a & 1)? printf("El numero es Impar") : printf("El numero es Par") )

int main()
{
        int num1;

        printf(''ingrese un numero \n'');
        scanf(''%d'',&num1);
        PAR(num1);

        return 0;
}
```

Otra herramienta útil de una macro función es lo que se denomina “redefinición” de una función, es decir, que puedo renombrar a una función y que haga exactamente lo mismo o con algunos ligeros cambios.

Supongamos que queremos redefinir la función “printf” de C y queremos que se llame IMPRIMIR y que siempre que se la cite imprima por pantalla “Mensaje: <texto que se desea imprimir>”. Entonces el código quedaría de la siguiente forma:

Redefinición de una Función

```
#include <stdio.h>
#define IMPRIMIR(s) \
                printf ("Mensaje: \"%s\"\n", s);

int main()
{
        IMPRIMIR(''Hola Mundo''); /*imprime por pantalla Mensaje: Hola Mundo*/
        return 0;
}
```

4.3. Otras Directivas del Preprocesador

4.3.1. #undef

La directiva #undef se usa para quitar una definición de nombre de macro que se haya definido previamente. Es decir que el objetivo principal de esta directiva es permitir localizar los nombres de macros sólo en las secciones de código que se necesiten.

uso de #undef

```
1  #include <stdio.h>
2  #define IMPRIMIR(s) \
3                  printf ("Mensaje: \"%s\"\n", s);
4
5  int main()
6  {
7          IMPRIMIR(''Hola Mundo''); /*imprime por pantalla Mensaje: Hola Mundo*/
8          #undef IMPRIMIR
9
10         /*si queremos ahora utilizar IMPRIMIR, dicha macro funcion fue ''eliminada'' con lo cual, no
11
12         IMPRIMIR(''funciona?''); /*verifiquen que esta linea va a producir un error al momento de co
13         return 0;
14 }
```

4.3.2. #if , #else , #elif

La directiva #if evalúa una expresión constante entera. Siempre se debe terminar con #endif para marcar el fin de esta sentencia.

Se pueden así mismo evaluar otro código en caso se cumpla otra condición, o bien, cuando no se cumple ninguna usando #else o #elif (se puede apreciar que es una combinación de #else e #if respectivamente).

uso de condicionales en macros

```
1  #include <stdio.h>
2
3  #define LUNES 0
4  #define MARTES 1
5
6
7  #define DIA_SEMANA LUNES
8
9  int main()
10 {
11         #if DIA_SEMANA == LUNES
12                 printf("lunes");
13         #elif DIA_SEMANA == MARTES
14                 printf("martes");
15         #else
16                 printf("sera fin de semana?");
17         #endif
18
19         return 0;
20 }
```

4.3.3. #error

La estructura básica de #error es:

- #error <mensaje_de_error>

Esta directiva forza al compilador a parar la compilación cuando la encuentra, mostrando el mensaje de error (<mensaje_de_error>) que se escribió. Se usa principalmente para depuración de errores.

uso de #error

```
1  #include <stdio.h>
2
3  #define DIVIDIR(a,b) (a/b)
4  #define NUM1 5
```

```
5  #define NUM2 0
6
7  int main()
8  {
9          #if NUM2 != 0
10                 printf("%f",DIVIDIR(NUM1,NUM2));
11         #else
12                 #error division por cero!
13         #endif
14         return 0;
15 }
```

Por ejemplo si usamos codeblocks para compilar dicho programa, nos mostraría un mensaje de error al compilar del estilo:

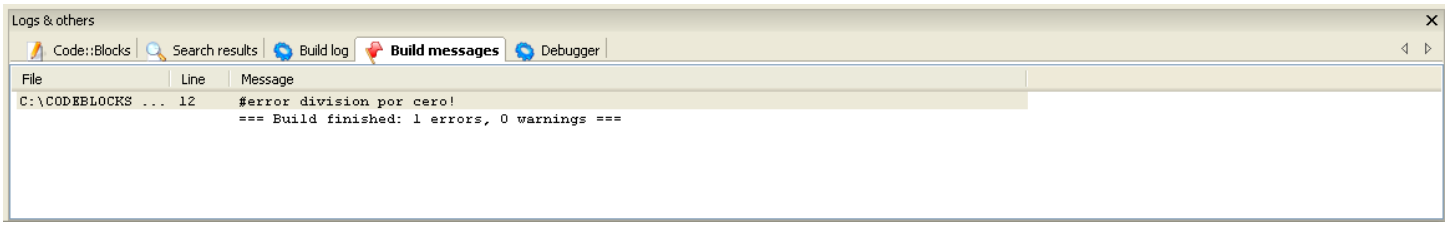

Figura 1: Mensaje de Error Utilizando #error

4.4. Ventajas y Desventajas de Usar Macros

La ventaja de usar macros son:

- Al utilizar Macro Funciones se acelera la ejecución del programa
- Al utilizar tanto Macro Funciones como Macro Objetos, las "variables locales" o "constantes" no ocupan espacio en memoria!!!

Pero como dice el dicho "todo lo bueno siempre tiene algo malo" los Macros poseen tres desventajas de gran consideración. Dichas desventajas son:

- El tamaño del código aumenta. Ya que se expande más el código y no hay manera de encapsularlo
- El uso de macros queda fuera del normal planteamiento estructurado de creación de programas. Es decir en tiempo de diseño no se suelen plantear macros, lo cual puede generar confusiones al momento de implementarlos y llevar una perdida de tiempo debatiendo si es conveniente o no construirlos
- Los lenguajes modernos (como C#, Java, etc) no utilizan macros, lo cual si tenemos que exportar el código hecho a alguno de esto lenguajes, vamos a tener que reconstruirlo y eso llevaría una pérdida de tiempo.