APÈNDIX: PREPROCESSADOR C

1. El preprocessador de C

La programació en C, especialment la de sistemes operatius, utilitza bastant les facilitats que dona els preprocessadors de C. Encara que se suposa que ja sabeu algunes directives bàsiques com `include` o `define`, aqui teniu una adreça web on podeu consultar un tutorial per utilitzar el preprocessador. Fixeu-vos que aquestes directives son també utilitzades en alguns fitxers assemblador (entry.S) per pre-processar-los abans de compilar-los. L'adreça és http://www.publispain.com/supertutoriales/programacion/c_y_cplus/cursos/4/Cpreproc.htm

2. Ayuda a programar con limpieza

El preprocesador es un programa que se ejecuta justa antes de la ejecución del compilador, su operación es transparente para Usted pero hace un trabajo muy importante al remover todos los comentarios del código fuente y efectuando una serie de sustituciones conceptuales basadas en su código pasando el resultado al compilador.

```
# include <stdio.h>

# define INICIO 0                          /* Punto de inicio del bucle
*/
# define FINAL 9                          /* Fin del bucle */
# define MAX(A,B) ((A)>(B)?(A):(B))       /* Definición macro de Max */
# define MIN(A,B) ((A)>(B)?(B):(A))       /* Definición macro de Min */

int main( )
{
    int indice, mn, mx ;
    int contador = 5 ;

    for (indice = INICIO ; indice <= FINAL ; indice++)
    {
        mx = MAX(indice, contador) ;
        mn = MIN(indice, contador) ;
        printf ( "Max es %d y min es %d\n", mx, mn) ;
    }
    return 0 ;
}

/* Resultado de la ejecución:
Max es 5 y min es 0
Max es 5 y min es 1
Max es 5 y min es 2
Max es 5 y min es 3
Max es 5 y min es 4
Max es 5 y min es 5
Max es 6 y min es 5
Max es 7 y min es 5
Max es 8 y min es 5
Max es 9 y min es 5
*/
```

Observe las líneas 3 a 6, cada una comienza con **#define**. Esta es la manera para declarar todas las macros y definiciones. Antes de iniciar el proceso de compilación, el compilador va a la etapa del preprocesador para resolver todas las definiciones, en el presente caso, se buscará cada lugar en el programa donde se encuentre la palabra **INICIO** y será reemplazada con un cero porque así está definido. El compilador en sí jamás verá la palabra **INICIO**. Observe que si la palabra se encuentra en una cadena o en un comentario, esta no será cambiada. Debe quedarle claro que al poner la palabra **INICIO** en lugar del número 0 es solo por conveniencia para Usted actuando como comentario ya que la palabra **INICIO** ayuda a entender el uso del cero.

Es una práctica común en la programación C utilizar letras mayúsculas para representar constantes simbólicas y utilizar letras minúsculas para los nombres de las variables. Usted puede utilizar el estilo de letra que más le guste ya que esto es materia de gusto personal.

2.1. ¿Qué es una macro?

Una macro no es otra cosa que una definición, pero como parece ser capaz de ejecutar algunas decisiones lógicas ú operaciones matemáticas, tiene un nombre único. En la línea 5 del programa podemos ver un ejemplo de una macro, en este caso, cada vez que el preprocesador encuentra la palabra **MAX** seguida por un grupo de paréntesis espera encontrar dos términos en el paréntesis y hará el reemplazo de los términos en la segunda parte de la definición, así el primer término reemplazará cada **A** en la segunda parte de la definición, y el segundo término reemplazará cada **B** en la segunda parte de la definición. Cuando el programa alcanza la línea 15, **indice** será sustituida por cada **A**, y **contador** será sustituida por cada **B**. Por lo tanto, antes de que la línea 15 sea entregada al compilador, esta será modificada por lo siguiente:

```
mx = ((index)>(count) ? (index) : (count))
```

Recuerde que ni los comentarios ni las cadenas serán afectados. Recordando las construcciones ya estudiadas vemos que **mx** recibirá el valor máximo de **indice** ó **contador**. De la misma manera, la macro **MIN** resulta en **mn** recibiendo el valor mínimo de **indice** ó **contador**. Estas dos macros se utilizan con frecuencia en los programas C. Al definir una macro es imperativo que no haya espacio entre el nombre de la macro y el paréntesis de apertura, de lo contrario, el compilador no podrá determinar la existencia de una macro pero sí hará la sustitución definida. Los resultados de la macro se imprimen en la línea 17.

2.2. Una macro equivocada

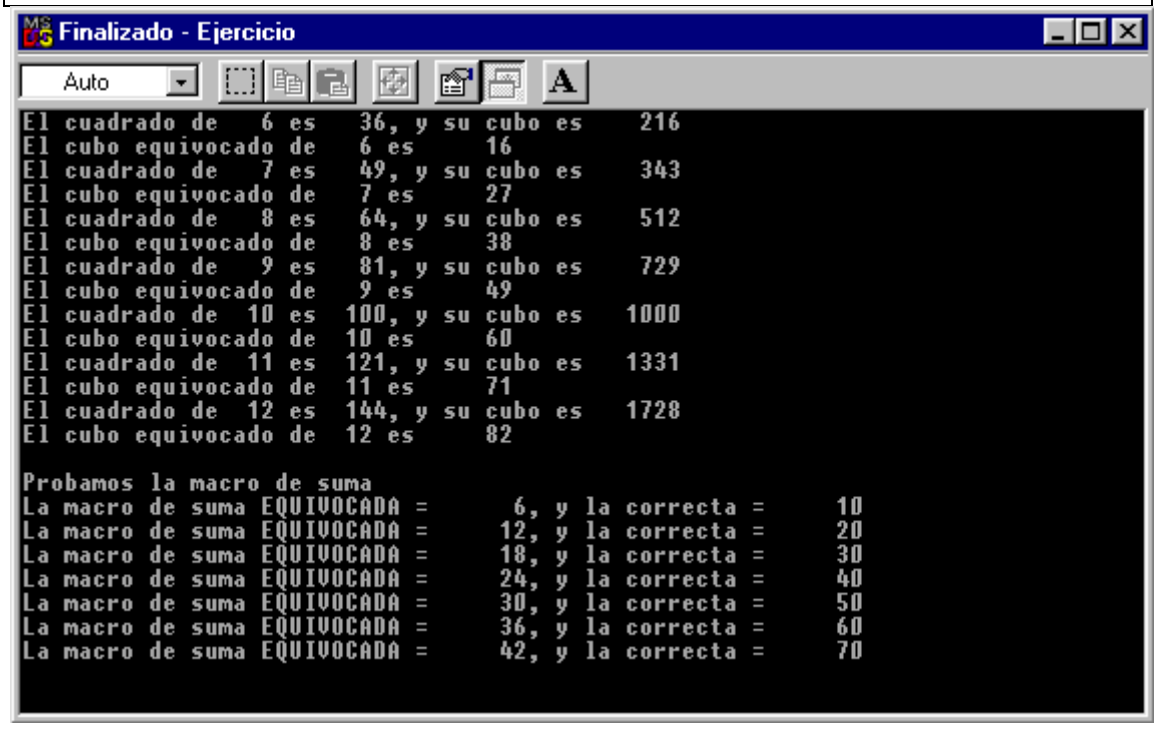
En el siguiente código podemos observar que la línea 3 define una macro llamada **EQUIVOCADA** que aparentemente calcula el cubo de **A**, y en algunos casos lo hace, pero falla miserablemente en otros casos. La segunda macro llamada **CUBO** obtiene el cubo pero no en todos los casos, mas adelante estudiaremos el porque falla en algunas situaciones, el código es el siguiente:

```
#include <stdio.h>

#define EQUIVOCADA(A) A*A*A              /* Macro EQUIVOCADA para el
cubo */
#define CUBO(A) (A)*(A)*(A)             /* Macro correcta para el cubo
*/
```

```
#define  CUADRADO(A)  (A)*(A)              /*  Macro  correcta  para  el
cuadrado */
#define  SUMA_EQUIVOCADA(A)  (A)+(A)      /*  Macro  equivocada  para  la
suma */
#define SUMA_CORRECTA(A) ((A)+(A))      /* Macro correcta para la suma
*/
#define INICIO 1
#define FINAL 7

int main( )
{
    int i, offset ;
    offset = 5 ;
    for (i = INICIO ; i <= FINAL ; i++)
    {
        printf ("El cuadrado de %3d es %4d, y su cubo es %6d\n",
        i+offset, CUADRADO(i+offset), CUBO(i+offset)) ;
        printf ("El cubo equivocado de %3d es %6d\n",
        i+offset, EQUIVOCADA(i+offset)) ;
    }
    printf ("\nProbamos la macro de suma\n") ;
    for (i = INICIO ; i <= FINAL ; i++)
    {
     printf  ("La  macro  de  suma  EQUIVOCADA  =  %6d,  y  la  correcta  =
%6d\n",
    5*SUMA_EQUIVOCADA(i), 5*SUMA_CORRECTA(i)) ;
    }
    return 0 ;
}
```



Considere el programa mismo donde el **CUBO** de **i+offset** se calcula en la línea 19. Si **i** es 1, entonces estaremos buscando el cubo de 1+5=6, lo cual resulta en 216. Cuando se usa **CUBO**, los valores se agrupan así,

(1+5)*(1+5)*(1+5)=6*6*6=216. Sin embargo, al utilizar **EQUIVOCADA** tenemos el siguiente agrupamiento, 1+5*1+5*1+5=1+5+5+5=16 lo que da un resultado erróneo. Los paréntesis son necesarios para agrupar adecuadamente las variables.

En la línea 6 definimos la macro **SUMA_EQUIVOCADA** de acuerdo a las reglas dadas pero aún tenemos problemas cuando tratamos de utilizar esta macro en las líneas 27 y 28. En la línea 28, cuando queremos que el programa calcule **5*SUMA_EQUIVOCADA(i)** con **i=1**, obtenemos como resultado 5*1+1, lo que se evalúa como 5+1 ó 6, y esto seguramente no es lo que tenemos en mente, el resultado que realmente deseamos es 5*(1+1) = 5*2 = 10 que es la respuesta que obtenemos al utilizar la macro llamada **SUMA_CORRECTA**, esto se debe a los paréntesis extra que agregamos en la definición dada en la línea 7.

Dedicarle un poco de tiempo para estudiar este programa nos ayudará a comprender el funcionamiento de las macros. Para prevenir los problemas que hemos visto en el ejemplo, los programadores experimentados de C incluyen un paréntesis en torno a cada variable en una macro y un paréntesis adicional en torno a la totalidad de la expresión, esto permitirá a cualquier macro trabajar adecuadamente y esta es la razón por la que la macro **CUBO** arroja ciertos resultados erróneos, necesita un paréntesis en torno a la expresión.

2.3. Compilación condicional (Parte 1).

Analicemos ahora el concepto de compilación condicional en el código siguiente. Se define **OPCION_1** en la línea 3, y se considera definida por el resto del programa, cuando el preprocesador alcanza la línea 5 mantiene el texto comprendido entre las líneas 5 y 7 en el programa y lo pasa al compilador. Si **OPCION_1** no hubiera sido definido en la línea 5, el preprocesador se hubiera brincado la línea 6 y el compilador jamás la hubiera visto. Similarmente la línea 17 es condicionalmente compilada siempre que **OPCION_1** lo sea. Esta es una construcción muy útil pero no en la manera en que la usamos en el ejemplo, generalmente se utiliza para incluir una característica si estamos utilizando cierto tipo de procesador, o cierto tipo de sistema operativo o aún una pieza especial de hardware.

```
#include <stdio.h>

#define OPCION_1       /* Esto define el control del preprocesador */
#ifdef OPCION_1
int contador_1 = 17;    /* Esto existe solo si OPCION_1 es definido */
#endif

int main( )
{
    int indice ;
    for (indice = 0 ; indice < 6 ; indice++)
    {
        printf ("En el bucle, indice = %d", indice) ;
        # ifdef OPCION_1
        printf (" contador_1 = %d", contador_1) ; /* puede desplegarse
*/
        # endif
        printf ("\n") ;
    }
    return 0 ;
}

# undef OPCION_1
```

```
/* Resultado de la ejecución:
(Con OPCION_1 definido)
En el bucle, indice = 0 contador_1 = 17
En el bucle, indice = 1 contador_1 = 17
En el bucle, indice = 2 contador_1 = 17
En el bucle, indice = 3 contador_1 = 17
En el bucle, indice = 4 contador_1 = 17
En el bucle, indice = 5 contador_1 = 17
(Comentando ó removiendo la línea 3)
En el bucle, indice = 0
En el bucle, indice = 1
En el bucle, indice = 2
En el bucle, indice = 3
En el bucle, indice = 4
En el bucle, indice = 5
*/
```

Compile y ejecute el programa como está, después comente la línea 3 de tal manera que **OPCION_1** no sea definida entonces recompile y ejecute el programa, verá como la línea extra no se imprimirá porque el preprocesador se la brincó.

En la línea 25 ilustramos el comando al preprocesador **undefine**. Este remueve el hecho de que **OPCION_1** fue definido y desde este punto el programa actúa como si nunca hubiera sido definido, por supuesto que la instrucción **undefine** nada tiene que hacer en este punto del programa ya que éste está completo y no siguen mas enunciados ejecutables, como experimento coloque la instrucción **undefine** en la línea 4, recompile y ejecute el programa y verá que actúa como si **OPCION_1** jamás hubiera sido definido.

2.4. Compilación condicional (Parte 2).

En el siguiente programa ilustramos la directiva al preprocesador **ifndef** que se lee literalmente "si no definido". El programa de ejemplo siguiente representa un ejercicio real de lógica para el estudiante diligente y no debe representar problema alguno comprender el uso de la instrucción **ifndef**.

```
# include <stdio.h>

#  define  OPCION_1                /*  Esto  define  el  control  al
preprocesador */
# define MUESTRA_DATO           /* Si es definido, se muestra*/
# ifndef OPCION_1
int  contador_1  =  17;            /*  Esto  existe  si  OPCION_1  no  es
definido */
# endif

int main( )
{
    int indice ;
    # ifndef MUESTRA_DATO
    printf ("MUESTRA_DATO no está definido en "
    " el codigo\n") ;
    # endif
    for (indice = 0 ; indice < 6 ; indice++)
    {
        # ifdef MUESTRA_DATO
        printf ("En el bucle, indice = %d", indice) ;
```

```
        # ifndef OPCION_1
         printf  ("  contador_1  =  %d",  contador_1);  /*  Esto  puede
mostrarse*/
        # endif
        printf ("\n") ;
        # endif
    }
    return 0 ;
}

/* Resultado de la ejecución:
(Con OPCION_1 definido)
En el bucle, indice = 0
En el bucle, indice = 1
En el bucle, indice = 2
En el bucle, indice = 3
En el bucle, indice = 4
En el bucle, indice = 5
(Removiendo ó comentando la línea 3)
En el bucle, indice = 0 contador_1 = 17
En el bucle, indice = 1 contador_1 = 17
En el bucle, indice = 2 contador_1 = 17
En el bucle, indice = 3 contador_1 = 17
En el bucle, indice = 4 contador_1 = 17
En el bucle, indice = 5 contador_1 = 17
*/
```

2.5. Compilación condicional (Parte 3).

El siguiente programa ilustra un uso práctico del preprocesador. En este programa definimos un símbolo llamado **EN_PROCESO**, cuando llegamos al código de la función main ( ) vemos el porque está definido. Aparentemente no tenemos suficiente información para completar este código por lo que decidimos separar el código hasta tener una oportunidad de hablar con Martín y Martha acerca de cómo completar estos cálculos, mientras tanto deseamos continuar trabajando en otras partes del programa por lo que utilizamos el preprocesador para temporalmente brincarnos esta parte incompatible del código, debido al mensaje que colocamos en la línea 14 es imposible olvidar que debemos regresar y limpiar el código. Veamos el ejemplo:

```
# include &ltstdio.h>

# define EN_PROCESO

int main( )
{
    int indice ;
    for (indice = 0 ; indice < 6 ; indice++)
    {
        printf ("Indice es ahora %d", indice) ;
        printf (" y podemos procesar los datos") ;
        printf ("\n") ;
        # ifdef EN_PROCESO
        printf ("El codigo no ha sido completado! *************\n") ;
        # else
        for (contador = 1 ; contador < indice * 5 ; contador++)
        {
            vale = (ver la pag. 16 de la documentación)
```

```
            limite = (Preguntar a Martin por este cálculo)
            Martha tiene una tabla de datos para el análisis del peor
caso
            printf ("contador = %d, vale = %d, limite = %d\n,
            contador, vale, limite) ;
         }
        # endif
    }
    return 0 ;
}

/* Resultado de la ejecución:
(Con EN_PROCESO definido)
Indice es ahora 0 y podemos procesar los datos.
El codigo no ha sido completado! *************
Indice es ahora 1 y podemos procesar los datos.
El codigo no ha sido completado! *************
Indice es ahora 2 y podemos procesar los datos.
El codigo no ha sido completado! *************
Indice es ahora 3 y podemos procesar los datos.
El codigo no ha sido completado! *************
Indice es ahora 4 y podemos procesar los datos.
El codigo no ha sido completado! *************
Indice es ahora 5 y podemos procesar los datos.
El codigo no ha sido completado! *************
(Removiendo ó comentando la línea 3)
(El programa no compilará por tener errores.)
*/
```

En este caso solo hemos tratado con unas cuantas líneas de código. Podemos utilizar esta técnica para manejar varios bloques de código, algunos de los cuales pueden estar en otros módulos, hasta que Martín regrese a explicar el análisis y así poder completar los bloques indefinidos.

2.6. Programas con múltiples archivos

Para programas pequeños es conveniente incluir todo el código en un solo archivo y compilarlo para obtener el resultado final, sin embargo, la gran mayoría de los programas C son muy grandes para incluirlos en un solo archivo y trabajar cómodamente. Es normal encontrar un programa compuesto de varios archivos y es necesario para estos archivos comunicarse y trabajar juntos en un solo programa grande. Aunque es mejor no utilizar variables globales, algunas veces es conveniente su uso. Algunas de estas variables necesitan ser referenciadas por dos o mas archivos diferentes, C provee una manera de hacer esto. Considere las siguientes tres porciones de código.

```
Archivo1.c                Archivo2.c              Archivo3.c
int indice;               extern int indice;      extern int indice;
extern int contador;      int contador;
                          Static int valor;       int valor;
int main ();
static void uno ();       void dos ();            void tres ();
```

La variable llamada **indice** definida en **Archivo1.c** está disponible para utilizarse por cualquier otro archivo porque está definida globalmente. Los otros dos archivos hacen uso de la misma variable al declararla variable de tipo **extern**. En esencia se le está diciendo al compilador, "deseo utilizar la variable llamada **indice** la cual está

definida en algún lugar". Cada vez que **indice** sea referenciada en los otros dos archivos, la variable de ese nombre es utilizada de **Archivo1.c**, y puede ser leída y modificada por cualquiera de los tres archivos, esto provee una manera fácil para intercambiar datos de un archivo a otro pero puede causar problemas.

La variable llamada **contador** esta definida en **Archivo2.c** y esta referida en **Archivo1.c** como explicamos arriba, pero no puede utilizarse en **Archivo3.c** porque aquí no está declarada. Una variable estática, como **valor** en **Archivo2.c** no puede ser referenciada por ningún otro archivo. Otra variable llamada **valor** está definida en **Archivo3.c**, esta no tiene ninguna relación con la variable del mismo nombre en **Archivo2.c**. En este caso, **Archivo1.c** puede declarar una variable externa **valor** y hacer referencia a esta variable en **Archivo3.c** si se desea.

El punto de entrada **main ( )** solo puede ser llamado por el sistema operativo para iniciar el programa, pero las funciones **dos ( )** y **tres ( )** pueden ser llamadas desde cualquier punto dentro de los tres archivos ya que son funciones globales. Sin embargo, como la función **uno ( )** esta declarada como de tipo estática solo puede ser llamada dentro del archivo en la cual esta declarada.

2.7. ¿Qué es una variable enumerada?

Veamos en el siguiente código un ejemplo de cómo utilizar la variable de tipo **enum**.

```
# include <stdio.h>

main()
{
    enum {CERO,UNO,DOS,TRES,CUATRO=15,CINCO}numero;

    numero=CERO;
     printf("La primera variable numero de tipo enum es: %d\n",
numero);
    numero=UNO;
     printf("La segunda variable numero de tipo enum es: %d\n",
numero);
    numero=DOS;
     printf("La tercera variable numero de tipo enum es: %d\n",
numero);
    numero=TRES;
    printf("La cuarta variable numero de tipo enum es: %d\n", numero);
    numero=CUATRO;
     printf("La quinta variable numero de tipo enum vale: %d\n",
numero);
    numero=CINCO;
    printf("La ultima variable numero de tipo enum es: %d\n", numero);
    return 0;
}
```