El preprocesador de C

Directivas del preprocesador

El preprocesador tiene más o menos su propio lenguaje el cual puede ser una herrramienta muy poderosa para el programador. Todas las directivas de preprocesador o comandos inician con un #.

Las ventajas que tiene usar el preprocesador son:

- los programas son más fáciles de desarrollar,
- son más fáciles de leer,
- son más fáciles de modificar
- y el código de C es más transportable entre diferentes arquitecturas de máquinas.

#define

El preprocesador también permite configurar el lenguaje. Por ejemplo, para cambiar a las sentencias de bloque de código { ... } delimitadores que haya inventado el programador como inicio ... fin se puede hacer:

```
#define inicio {
#define fin }
```

Durante la compilación todas las ocurrencias de inicio y fin serán reemplazadas por su correspondiente { o } delimitador y las siguientes étapas de compilación de C no encontrarán ninguna diferencia.

La directiva #define se usa para definir constantes o cualquier sustitución de macro. Su formato es el siguiente:

#define <nombre de macro> <nombre de reemplazo>

Por ejemplo:

```
#define FALSO 0
#define VERDADERO !FALSO
```

La directiva #define tiene otra poderosa característica: el nombre de macro puede tener argumentos. Cada vez que el compilador encuentra el nombre de macro, los argumentos reales encontrados en el programa reemplazan los argumentos asociados con el nombre de la macro. Por ejemplo:

```
#define MIN(a,b) (a < b) ? a : b

main()
{
        int x=10, y=20;
        printf("EL minimo es %d\n", MIN(x,y) );
}
```

Cuando se compila este programa, el compilador sustituirá la expresión definida por MIN(x,y), excepto que x e y serán usados como los operandos. Así después de que el compilador hace la sustitución, la sentencia printf será ésta:

```
printf("El minimo es %d\n", (x < y) ? x : y);
```

Como se puede observar donde se coloque MIN, el texto será reemplazado por la definición apropiada. Por lo tanto, si en el código se hubiera puesto algo como:

```
x = MIN(q+r,s+t);
```

después del preprocesamiento, el código p odría verse de la siguiente forma:

```
x = ( q+r < s+t ) ? q+r : s+t;
```

Otros ejemplos usando #define pueden ser:

```
#define Deg_a_Rad(X) (X*M_PI/180.0)
/* Convierte grados sexagesimales a radianes, M_PI es el valor de pi */
/*    y esta definida en la bibliotec a math.h */

#define IZQ_DESP_8 <<8
```

La última macro IZQ_DESP_8 es solamente válida en tanto el reemplazo del contexto es válido, por ejemplo: x = y IZQ_DESP_8.

El uso de la sustitución de macros en el lugar de las funciones reales tiene un beneficio impor tante: incrementa la velocidad del código porque no se penaliza con una llamada de función. Sin embargo, se paga este incremento de velocidad con un incremento en el tamaño del programa porque se duplica el código.

#undef

Se usa #undef para quitar una definición de nombre de macro que se haya definido previamente. El formato general es:

#undef <nombre de macro>

El uso principal de #undef es permitir localizar los nombres de macros sólo en las secciones de código que los necesiten.

#include

La directiva del preprocesador **#include** instruye al compilador para incluir otro archivo fuente que esta dado con esta directiva y de esta forma compilar otro archivo fuente. El archivo fuente que se leerá se debe encerrar entre comillas dobles o paréntesis de ángulo . Por ejemplo:

```
#include <archivo>

#include "archivo"
```

Cuando se indica <archivo> se le dice al compilador que busque donde estan los archivos incluidos o ``include" del sistema. Usualmente los sistemas con UNIX guardan los archivos en el directorio /usr/include.

Si se usa la forma "archivo" es buscado en el directorio actual, es decir, donde el programa esta siendo ejecutado.

Los archivos **incluidos** usualmente contienen los prototipos de las funciones y las declaraciones de los archivos cabecera (header files) y no tienen código de C (algoritmos).

#if Inclusión condicional

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

Se pueden así mismo evaluar otro código en caso s e cumpla otra condición, o bien, cuando no se cumple ninguna usando #elif o #else respectivamente.

Por ejemplo,

```
#define MEX 0
#define EUA 1
#define FRAN 2
#define PAIS_ACTIVO MEX

#if PAIS_ACTIVO == MEX
        char moneda[]="pesos";
#elif PAIS_ACTIVO == EUA
        char moneda[]="dolar";
#else
        char moneda[]="franco";
#endif
```

Otro método de compilación condicional usa las directivas **#ifdef** (si definido) y **#ifndef** (si no definido).

El formato general de **#ifdef** es:

#ifdef <nombre de macro>
<secuencia de sentecias>
#endif

Si el nombre de macro ha sido definido en una sentencia **#define**, se compilará la secuencia de sentecias entre el **#ifdef** y **#endif**.

El formato general de **#ifdef** es:

#ifndef <nombre de macro>
<secuencia de sentecias>
#endif

Las directivas anteriores son útiles para revisar si las macros están definidas -- tal vez por módulos diferentes o archivos de cabecera.

Por ejemplo, para poner el tamaño de un entero para un programa portable entre TurboC de DOS y un sistema operativo con UNIX, sabiendo que T urboC usa enteros de 16 bits y UNIX enteros de 32 bits, entonces si se quiere compilar para TurboC se puede definir una macro TURBOC, la cual será usada de la siguiente forma:

```
#ifdef TURBOC
#define INT_SIZE 16
#else
#define INT_SIZE 32
#endif
```

Control del preprocesador del compilador

Se puede usar el compilador para controlar los valores dados o definidos en la línea de comandos. Esto permite alguna flexibilidad para configurar valores ademas de tener algunas otras funciones útiles. Por ejemplo cuando se quiere como una bandera para depuración se puede hacer algo como lo siguiente:

```
#ifdef DEBUG
        printf("Depurando: Versión del programa 1.0 \n");
#else
        printf("Version del programa 1.0 (Estable) \n");
#endif
```

Como los comandos del preprocesador pueden estar en cualquier parte de un programa, se pueden filtrar variables para mostrar su valor, cuando se esta depurando, ver el siguiente ejemplo:

```
x = y * 3;

#ifdef DEBUG
   printf("Depurando: variables x e y iguales a  \n",x,y);
#endif
```

Otras directivas del preprocesador

La directiva **#error** forza al compilador a parar la compilación cuando la encuentra. Se usa principalmente para depuración. Por ejemplo:

```
#ifdef OS_MSDOS
   #include <msdos.h>
#elifdef OS_UNIX
   #include "default.h"
#else
   #error Sistema Operativo incorrecto
#endif
```

La directiva **#line** número "cadena"' informa al preprocesador cual es el número siguiente de la línea de entrada. Cadena es opcional y nombra la siguiente línea de entrada. Esto es usado frecuentemente cuando son traducidos otros lenguajes a C. Por ejemplo, los mensajes de error producidos por el compilador de C pueden referenciar el nombre del archivo y la línea de la fuente original en vez de los archivos intermedios de C. Por ejemplo, lo siguiente especifica que el contador de línea empezará con 100.

```
#line 100 "test.c"    /* inicializa el contador de linea y nombre de archivo */
main()                /* linea 100 */
{
   printf("%d\n",__LINE__); /* macro predefinida, linea 102 */
   printf("%s\n",__FILE__); /* macro predefinida para el nombre */
}
```

Sentencias

Las ***expresiones*** de C son unidades o componentes elementales de unas entidades de rango superior que son las ***sentencias***. Las sentencias son unidades completas, ejecutables en sí mismas. Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

Sentencias Simples

Una sentencia simple es una expresión de algún tipo terminada con un carácter (;). Un caso típico son las declaraciones o las sentencias aritméticas. Por ejemplo:

```
float real;
espacio = espacio_inicial + velocidad * tiempo;
```

Sentencia Vacía o Nula

En algunas ocasiones es necesario introducir en el programa una sentencia que ocupe un lugar, pero que no realice ninguna tarea. A esta sentencia se le denomina sentencia vacía y consta de un simple carácter (;). Por ejemplo:

```
;
```

Sentencias Compuestas O Bloques

Muchas veces es necesario poner varias sentencias en un lugar del programa donde debería haber una sola. Esto se realiza por medio de sentencias compuestas. Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves {...}. También se conocen con el nombre de bloques. Una sentencia compuesta puede incluir otras sentencias, simples y com puestas. Un ejemplo de sentencia compuesta es el siguiente:

```
{
int i = 1, j = 3, k;
double masa;
masa = 3.0;
k = y + j;
}
```

Estructuras Condicionales

El control de flujo lógico de un programa se puede realizar mediante varios métodos. Pero antes recordaremos los operadores relaciones binarios que se utilizan:

==, !=, <, <=, > y >=

además los operadores lógicos binarios:

||, &&

y el operador lógico unario de negación !, que sólo toma un argumento. Los operadores son utilizados con las siguientes estructu ras.

La sentencia if

Las tres formas como se puede emplear la sentencia if son:

```
if (condicion)
sentencia;
```

...o

```
                                        if (condicion)
                                        sentencia1;
                                        else
                                        sentencia2;
...o
                                        if (condicion1)
                                        sentencia1;
                                        else if (condicion2)
                                        sentencia2;
                                        ...
                                        else
                                        sentencian;
```

El flujo lógico de esta estructura es de arriba hacia abajo.

La primera sentencia se ejecutará y se saldrá de la estructura *if* si la primera condición es verdadera.

Si la primera condición fue falsa, y existe otra condición, se evalúa, y si la c ondición es verdadera, entonces se ejecuta la sentencia asociada.

Si existen más condiciones dentro de la estructura if, se van evaluando éstas, siempre y cuando las condiciones que le precedan sean falsas.

La sentencia que esta asociada a la palabra r eservada *else*, se ejecuta si todas las condiciones de la estructura *if* fueron falsas. Por ejemplo:

```
main()
{
  int x, y, w;
        ........
  if (x>0)
  {
    z=w;
    .......
  }
  else
  {
    z=y;
    .......
  }
}
```

El operador ?

El operador ternario condicional ? es más eficiente que la sentencia if. El operador ? tiene el siguiente formato:

expresion1 ? expresion 2 : expresion3;

Que es equivalente a la siguiente expresión:

if (expresion1) then expresion2 else expresion3;

Por ejemplo, para asignar el máximo de a y b a la variable z, usando ?, tendríamos:

z = (a>b) ? a : b;

que es lo mismo que:

```
if (a > b)
   z = a;
else
   z = b;
```

El uso del operador ? para reemplazar las sentencias if ... else no se restringe sól o a asignaciones, como en el ejemplo anterior. Se pueden ejecutar una o más llamadas de función usando el operador ? poniéndolas en las expresiones que forman los operandos, como en el ejemplo siguiente:

```
f1(int n)
{
   printf("%d ",n);
}

f2()
{
   printf("introducido\n");
}

main()
{
   int t;

   printf(": ");
   scanf("%d",&t);

   /* imprime mensaje apropiado */
   t ? f1(t) + f2() : printf("Se dió un cero \n");

}
```

La sentencia switch

Aunque con la estructura if ... else if se pueden realizar comprobaciones múltiples, en ocasiones no es muy elegante, ya que el código puede ser difícil de seguir y puede confundir incluso al autor transcurrido un tiempo. Por lo anterior, C tiene incorporada una sentencia de bifurcación múltiple llamada *switch*.

Con esta sentencia, la computadora comprueba una variable sucesivamente frente a una lista de constantes enteras o de caracter. Después de encontrar una coincidencia, la computadora ejecuta la sentencia o bloque de sentencias que se asocian con la constant e. La forma general de la sentencia switch es:

```
switch (variable) {
case constante1:
secuencia de sentencias
break;
case constante2:
secuencia de sentencias
break;
case constante3:
```

```
secuencia de sentencias
break;
...
default:
secuencia de sentencias
}
```

donde la computadora ejecuta la sentencia default si no coincide ninguna constante con la variable, esta última es opcional. Cuando se encuentra una coincidencia, la computadora ejecuta las sentencias asociadas con el *case* hasta encontrar la sentencia break con lo que sale de la estructura switch.

Las limitaciones que tiene la sentencia switch ... case respecto a la estructura if son:

- Sólo se tiene posibilidad de revisar una sola variable.
- Con switch sólo se puede comprobar por igualdad, mien tras que con if puede ser con cualquier operador relacional.
- No se puede probar más de una constante por case.

La forma como se puede simular el último punto, es no teniendo sentencias asociados a un *case*, es decir, teniendo una sentencia nula donde sólo se pone el caso, con lo que se permite que el flujo del programa caiga al omitir las sentencias, como se muestra a continuación:

```
switch (letra)
{
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    numvocales++;
    break;

  case ' ':
    numesp++;
    break;

  default:
    numotras++;
    break;
}
```

Iteración

Mecanismo para repetir un conjunto de instrucciones hasta que se cumple cierta condición.

La sentencia for

La sentencia for tiene el siguiente formato:
for (expresion1; expresion2; expresion3)
sentencia;
o { bloque de sentencias }

En donde expresion1 se usa para realizar la inicialización de variables, usando una o varias sentencias, si se usan varias sentencias deberá usarse el operador , para separarlas. Po r lo general, establece el valor de la variable de control del ciclo. expresion2 se usa para la condición de

terminación del ciclo y expresion3 es el modificador a la variable de control del ciclo cada vez que la computadora lo repite, pero también puede s er más que un incremento. Por ejemplo:

```
int X;

main()
{
   for( X=3; X>0; X--)
   {
      printf("X=%d\n",X);
   }
}
```

genera la siguiente salida a pantalla ...

```
X=3
X=2
X=1
```

Todos las siguientes sentencias for son válidas en C:

```
for ( x=0; ( (x>3) && (x<9) ); x++ )
for ( x=0, y=4; ( (x>3) && (x<9) ); x++, y+=2)
for ( x=0, y=4, z=4000; z; z/=10)
```

En el segundo ejemplo se muestra la forma como múltiples expresiones pueden aparecer, siempre y cuando estén separadas por una coma , En el tercer ejemplo, el ciclo continuará iterando hasta que z se convierta en 0.

La sentencia while

La sentencia while es otro ciclo o bucle disponible en C. Su formato es:

while (expresion) sentencia;

donde sentencia puede ser una sentencia vacía, una sentencia única o un bloque de sentencias que se repetirán. Cuando el flujo del programa llega a esta instrucción, primero se revisa si la condición es verdad para ejecutar la(s) sentencia(s), y después el ciclo while se repetirá mientras la condición sea verdadera. Cuando lle ga a ser falsa, el control del programa pasa a la línea que sigue al ciclo. En el siguiente ejemplo se muetra una rutina de entrada desde el teclado, la cual se cicla mientras no se pulse A:

```
main()
{
   char carac;

   carac = '\0';
   while( carac != 'A') carac = getchar();
}
```

Antes de entrar al ciclo se inicializa la variable carac a nulo. Después pasa a la sentencia while donde se comprueba si carac no es igual a 'A', como sea verdad entonces se ejecuta la sentencia del bucle (carac = getchar();). La función getchar() lee el siguiente carácter del flujo estándar (teclado) y lo devuelve, que en nuestro ejemplo es el caracter que haya sido tecleado. Una vez que se ha pulsado una tecla, se asigna a carac y se comprueba la condición nuevamente. Después de pulsar A, la condición llega a ser falsa porque carac es igual a A, con lo que el ciclo termina.

De lo anterior, se tiene que tanto el ciclo for, como el ciclo while comprueban la condición en lo alto del ciclo, por lo que el código dentro del ciclo no se ejecuta siempre. Ejemplo:

```
main()
{
   int x=3;

   while( x>0 )
   {
      printf("x = %d\n", x);
      x--;
   }
}
```

que genera la siguiente salida en pantalla:

```
x = 3
x = 2
x = 1
```

Como se observa, dentro del ciclo tenemos más de una sentencia, po r lo que se requiere usar la llave abierta y la llave cerrada { ... } para que el grupo de sentencias sean tratadas como una unidad. Como el ciclo while pueda aceptar también expresiones, y no solamente condiciones lo siguiente es válido:

```
while ( x-- );
while ( x = x + 1 );
while ( x += 5 );
```

Si se usan este tipo de expresiones, solamente cuando el resultado de x --, x=x+1 o x+=5 sea cero, la condición fallará y se podrá salir del ciclo. De acuerdo a lo anterior, podemos realizar una operación completa dentro de la expresión. Por ejemplo:

```
main()
{
   char carac;

   carac = '\0';
   while ( (carac = getchar()) != 'A' )
      putchar(carac);
}
```

En este ejemplo se usan las funciones de la biblioteca estándar getchar() -- lee un caracter del teclado y putchar() escribe un caracter dado en pantalla. El ciclo while procederá a leer del teclado y lo mostrará hasta que el caracter A sea leído.

La sentencia do-while

Al contrario de los ciclos for y while que comprueban la condición en lo alto del bucle, el bucle do ... while la examina en la parte baja del mismo. Esta característica provoca que un ciclo do ... while siempre se ejecute al menos una vez. La forma general del ciclo es:

```
do {
sentencia;
} while (condición);
```

Aunque no son necesarias las llaves cuando sólo está presente una sentencia, se usan normalmente por legibilidad y para evitar confusión (respecto al lector, y no del compilador) con la sentencia while. En el siguiente programa se usa un ciclo do ... while para leer números desde el teclado hasta que uno de ellos es menor que o igual a 100:

```
main()
{
   int num;

   do
   {
      scanf("%d", &num);
   } while ( num>100 );
}
```

Otro uso común de la estructura do ... while es una rutina de selección en un menú, ya que siempre se requiere que se ejecute al menos una vez.

```
main()
{
   int opc;

   printf("1. Derivadas\n");
   printf("2. Limites\n");
   printf("3. Integrales\n");

   do
   {
      printf("   Teclear una opcion:  ");
      scanf("%d", &opc);

      switch(opc)
      {
         case 1:
            printf("\tOpcion 1
seleccionada\n\n");
            break;
         case 2:
            printf("\tOpcion 2
seleccionada\n\n");
            break;
         case 3:
            printf("\tOpcion 3
seleccionada\n\n");
            break;
         default:
            printf("\tOpcion no
disponible\n\n");
            break;
      }
   } while( opc != 1  &&  opc != 2  &&  opc
!= 3);
}
```

Ejemplo donde se reescribe usando do ... while uno de los ejemplos ya mostrados.

```
main()
{
   int x=3;

   do
   {
      printf("x = %d\n", x--);
   }
   while( x>0 ) ;
}
```

Uso de break y continue

Uno de los usos de la sentencia break es terminar un case en la sentencia switch. Otro uso es forzar la terminación inmediate de un ciclo, saltando la prueba condicional del ciclo. Cuando se encuentra la sentencia break en un bucle, la computadora termina inmediatamente el ciclo y el control del programa pasa a la siguiente sentecia del ciclo. Por ejemplo:

```
main()
{
   int t;
```

```
    for(t=0; t<100; t++)
    {
       printf("%d ", t);
       if (t==10) break;
    }
}
```

Este programa muestra en pantalla los números del 0 al 10, cuando alcanza el valor 10 se cumple la condición de la sentencia if, se ejecuta la sentencia break y sale del ciclo. La sentencia continue funciona de manera similar a la sentencia break. Sin embargo, en vez de forzar la salida, continue fuerza la siguiente iteración, por lo que salta el código que falta para llegar a proba r la condición. Por ejemplo, el siguiente programa visualizará sólo los números pares:

```
main()
{
   int x;

   for( x=0; x<100; x++)
   {
      if (x%2)
         continue;
      printf("%d ",x);
   }
}
```

Finalmente se considera el siguiente e jemplo donde se leen valores enteros y se procesan de acuerdo a las siguientes condiciones. Si el valor que sea leído es negativo, se desea imprimir un mensaje de error y se abandona el ciclo. Si el valor es mayor que 100, se ignora y se continua leyendo, y si el valor es cero, se desea terminar el ciclo.

```
main()
{
   int valor;

   while( scanf("%d", &valor) == 1 && valor
!= 0)
   {
      if ( valor<0 )
      {
         printf("Valor no valido\n");
         break;
         /* Salir del ciclo */
      }
      if ( valor>100)
      {
         printf("Valor no valido\n");
         continue;
         /* Pasar al principio del ciclo
nuevamente */
      }

      printf("Se garantiza que el valor leido
esta entre 1 y 100");
   }
}
```

FUNCIONES

Una ***función*** es una parte de código independiente del programa principal y de otras funciones, que puede ser llamada enviándole unos datos (o sin enviarle nada), para que realice una determinada tarea y/o proporcione unos resultados.

Utilidad de las funciones

Parte esencial del correcto diseño de un programa de computadora es su modularidad, esto es su división en partes más pequeñas de finalidad muy concreta.

En C estas partes de código reciben el nombre de funciones.

Las funciones facilitan el desarrollo y mantenimiento de los programas, evitan errores, ahorran memoria y trabajo innecesario.

Una misma función puede ser utilizada por diferentes programas, y por tanto no es necesario reescribirla. Además, una función es una parte de código independiente del programa principal y de otras funciones, manteniendo también independencia entre las variables respectivas y evitando errores y otros efectos colaterales de las modificaciones que se introduzcan.

Mediante el uso de funciones se consigue un código limpio, claro y elegante. La adecuada división de un programa en funciones constituye un aspecto fundamental en el desarrollo de programas de cualquier tipo. Las funciones, ya compiladas, pueden guardarse en librerías. Las librerías s on conjuntos de funciones compiladas, normalmente con una finalidad análoga o relacionada, que se guardan bajo un determinado nombre listas para ser utilizadas por cualquier usuario.

Definición de una función

La definición de una función consiste en la definición del código necesario para que ésta realice las tareas para las que ha sido prevista y se debe realizar en alguno de los archivos que forman parte del programa. La forma general de la definición de una función es la siguiente:

```
tipo_valor_de_retorno nombre_funcion(lista de argumentos con tipos)

{
declaración de variables y/o de otras funciones
codigo ejecutable
return (expresión); // optativo
}
```

La primera línea recibe el nombre de encabezamiento (header) y el resto de la definición –encerrado entre llaves– es el cuerpo (body) de la función.

Cada función puede disponer de sus propias variables, declaradas al comienzo de su código.

Estas variables, por defecto, son de tipo auto, es decir, sólo son visibles dentro del bloque en el que han sido definidas, se crean cada vez que se ejecuta la función y permanecen ocultas para el resto del programa.

Si estas variables de definen como static, conservan su valor entre distintas llamadas a la función.

También pueden hacerse visibles a la función variables globales definidas en otro archivo (o en el mismo archivo, si la definición está por debajo de donde se utilizan), declarándolas con la palabra reservada extern.

El código ejecutable es el conjunto de instrucciones que deben ejecutarse cada vez que la función es llamada.

La lista de argumentos con tipos, también llamados argumentos formales, es una lista de declaraciones de variables, precedidas por su tipo correspondiente y separadas por comas (,).

Los argumentos formales son la forma más natural y directa para que la función reciba valores desde el programa que la llama, correspondiéndose en número y tipo con otra lista de argumentos

-los argumentos actuales- en el programa que realiza la llamada a la función. Los argumentos formales son declarados en el encabezamiento de la función, pero no pueden ser inicializados en él.

Cuando una función es ejecutada, puede devolver al programa que la ha llamado un valor (el valor de retorno), cuyo tipo debe ser especificado en el encabezamiento de la funci ón (si no se especifica, se supone por defecto el tipo int). Si no se desea que la función devuelva ningún valor, el tipo del valor de retorno deberá ser void.

La sentencia return permite devolver el control al programa que llama. Puede haber varias sentencias return en una misma función. Si no hay ningún return, el control se devuelve cuando se llega al final del cuerpo de la función. Le palabra clave return puede ir seguida de una expresión, en cuyo caso ésta es evaluada y el valor resultante devuelto al programa que llama como valor de retorno (si hace falta, con una conversión previa al tipo declarado en el encabezamiento). Los paréntesis que engloban a la expresión que sigue a return son optativos.

El valor de retorno es un valor único: no puede ser un vector o una matriz, (temas que veremos más adelante) aunque sí un puntero a un vector o a una matriz. Sin embargo, el valor de retorno sí puede ser una estructura, que a su vez puede contener vectores y matrices como elementos miembros.

Como ejemplo supóngase que se va a calcular el valor absoluto de variables de tipo double. Una solución es definir una función que reciba como argumento el valor de la variable y devuelva ese valor absoluto como valor de retorno. La definición de esta función podría ser como sigue:

```
double valor_abs(double x)

{
if (x < 0.0)
return -x;
else
return x;
}
```

Declaración y llamada de una función

De la misma manera que en C es necesario declarar todas las variables, también toda función debe ser declarada antes de ser utilizada en la función o programa que realiza la llamada. De todas formas, ahora se verá que aquí hay una mayor flexibilidad que en el caso de las variables.

En C la declaración de una función se puede hacer de tres maneras:

a) Mediante una llamada a la función. En efecto, cuando una función es llamada sin que previamente haya sido declarada o definida, esa llamada sirve como declaración suponiendo int como tipo del valor de retorno, y el tipo de los argumentos actuales
como tipo de los argumentos formales. Esta práctica es muy peligrosa (es fuente de numerosos errores) y debe ser evitada.

b) Mediante una definición previa de la función. Esta práctica es segura si la definición precede a la llamada, pero tiene el inconveniente de que si la definición se cambia de lugar, la propia llamada pasa a ser declaración como en el caso a).

c) Mediante una declaración explícita, previa a la llamada. Esta es la práctica más segura y la que hay que tratar de seguir siempre. La declaración de la función se hace mediante el pro totipo de la función, bien fuera de cualquier bloque, bien en la parte de
declaraciones de un bloque.

C++ es un poco más restrictivo que C, y obliga a declarar explícitamente una función antes de llamarla.

La forma general del prototipo de una función es la siguiente:

tipo_valor_de_retorno nombre_funcion(lista de tipos de argumentos);

Esta forma general coincide sustancialmente con la primera línea de la definición –el encabezamiento-, con dos pequeñas diferencias: en vez de la lista de argumentos formales o parámetros, en el prototipo basta incluir los tipos de dichos argumentos. Se pueden incluir también identificadores a continuación de los tipos, pero son ignorados por el compilador.

Además, una segunda diferencia es que el prototipo termina con un carácter (;). Cuando no hay argumentos formales, se pone entre los paréntesis la palabra void, y se pone también void precediendo al nombre de la función cuando no hay valor de retorno.

Los prototipos permiten que el compilador realice correctamente la conversión del tipo del valor de retorno, y de los argumentos actuales a los tipos de los argumentos formales. La declaración de las funciones mediante los prototipos suele hacerse al comienzo del archivo, después de los #define e #include.

En muchos casos –particularmente en programas grandes, con muchos archivos y muchas funciones –, se puede crear un archivo (con la extensión .h) con todos los prototipos de las funciones utilizadas en un programa, e incluirlo con un #include en todos los archivos en que se utilicen dichas funciones.

La llamada a una función se hace incluyendo su nombre en una expresión o sentencia del programa principal o de otra función. Este nombre debe ir seguido de una lista de argumentos separados por comas y encerrados entre paréntesis. A los argumentos incluidos en la llamada se les llama argumentos actuales, y pueden ser no sólo variables y/o constantes, sino también expresiones.

Cuando el programa que llama encuentra el nombre de la función, evalúa los argumentos actuales contenidos en la llamada, los convierte si es necesario al tipo de los argumentos formales, y pasa copias de dichos valores a la función junto con el control de la ejecución.

El número de argumentos actuales en la llamada a una función debe coincidir con el número de argumentos formales en la definición y en la declaración. Existe la posibilidad de definir funciones con un número variable o indeterminado de argumentos.
Este número se concreta luego en el momento de llamarlas. Las funciones printf() y scanf() son ejemplos de funciones con número variable de argumentos.

Cuando se llama a una función, después de realizar la conversión de los argumentos actuales, se ejecuta el código correspondiente a la función hasta que se llega a una sentencia return o al final del cuerpo de la función, y entonces se devuelve el control al programa que realizó la llamada, junto con el valor de retorno si es que existe (convertido previamente al tipo especificado en el prototipo, si es necesario).

Recuérdese que el valor de retorno puede ser un valor numérico, una dirección (un puntero), o una estructura, pero no una matriz o un vector.

La llamada a una función puede hacerse de muchas formas, dependiendo de qué clase de tarea realice la función. Si su papel fundamental es calcular un valor de retorno a partir de uno o más argumentos, lo más normal es que sea llamada incluyendo su nombre seguido de los argumentos actuales en una expresión aritmética o de otro tipo.

En este caso, la llamada a la función hace el papel de un opera ndo más de la expresión. Obsérvese cómo se llama a la función seno en el ejemplo siguiente:

```
a = d * sin(alpha) / 2.0;
```

En otros casos, no existirá valor de retorno y la llamada a la función se hará incluyendo en el programa una sentencia que contenga sola mente el nombre de la función, siempre seguido por los argumentos actuales entre paréntesis y terminando con un carácter (;).

Por ejemplo, la siguiente sentencia llama a una función que multiplica dos matrices (nxn) A y B, y almacena el resultado en otra matriz C. Obsérvese que en este caso no hay valor de retorno (un poco más adelante se trata con detalle la forma de pasar vectores y matrices como argumentos de una función):

```
prod_mat(n, A, B, C);
```

Hay también casos intermedios entre los dos anteriores, c omo sucede por ejemplo con las funciones de entrada/. Dichas funciones tienen valor de retorno, relacionado de ordinario con el número de datos leídos o escritos sin errores, pero es muy frecuente que no se haga uso de dicho valor y que se llamen al modo de las funciones que no lo tienen.

La declaración y la llamada de la función valor_abs() antes definida, se podría realizar de la forma siguiente. Supóngase que se crea un archivo prueba.c con el siguiente contenido:

```
// archivo prueba.c
#include <stdio.h>

double valor_abs(double); // declaración

void main (void)
{
double z, y;
y = -30.8;
z = valor_abs(y) + y*y; // llamada en una expresion
}
```

La función valor_abs() recibe un valor de tipo double. El valor de retorno de dicha función (el valor absoluto de y), es introducido en la expresión aritmética que calcula z.

La declaración (double valor_abs(double)) no es estrictamente necesaria cuando la definición de la función está en el mismo archivo buscar.c que main(), y dicha definición está
antes de la llamada.

Se examina a continuación un ejemplo que encuentra el promedio de dos enteros:

```
float encontprom(int num1, int num2)
{
   float promedio;

   promedio = (num1 + num2) / 2.0;
   return(promedio);
}

main()
{
   int a=7, b=10;
```

```
    float resultado;

    resultado = encontprom(a, b);
    printf("Promedio=%f\n",resultado);
}
```

Funciones void

Las funciones void dan una forma de emular, lo que en otros lenguajes se conocen como procedimientos (por ejemplo, en PASCAL). Se usan cuando no requiere regres ar un valor. Se muestra un ejemplo que imprime los cuadrados de ciertos números.

```
void cuadrados()
{
    int contador;

    for( contador=1; contador<10; contador++)
        printf("%d\n",contador*contador);
}

main()
{
    cuadrados();
}
```

En la función cuadrados no esta definido ningún parámetro y por otra parte tampoco se emplea la sentencia return para regresar de la función.

Funciones para cadenas de caracteres

En C, existen varias funciones útiles para el manejo de cadenas de caracteres. Las más ut ilizadas son: strlen(), strcat(), strcmp() y strcpy(). Sus prototipos o declaraciones están en el archivo string.h, y son los siguientes:

Función Strlen()

El prototipo de esta función es como sigue:

```
unsigned strlen(const char *s);
```

Explicación: Su nombre proviene de string length, y su misión es contar el número de caracteres de una cadena, sin incluir el '\0' final. El paso del argumento se realiza por referencia, pues como argumento se emplea un puntero a la cadena (tal que el valor al que apunta es constante para la función; es decir, ésta no lo puede modificar), y devuelve un entero sin signo que es el número de caracteres de la cadena. La palabra const impide que dentro de la función lac cadena de caracteres que se pasacomo argumento sea modificada .

Función Strcat()

El prototipo de esta función es como sigue:

```
char *strcat(char *s1, const char *s2);
```

Explicación: Su nombre proviene de string concatenation y se emplea para unir dos cadenas de caracteres poniendo s2 a continuación de s1. El valor de retorno es un puntero a s1. Los argumentos

son los punteros a las dos cadenas que se desea unir. La función almacena la cadena completa en la primera de las cadenas. ¡PRECAUCIÓN! Esta función no prevé si tienesitio suficiente para almacenar las dos cadenas juntas en el espacio reservado para la primera. Esto es responsabilidad del programador.

Funciones Strcmp() Y Strcomp()

El prototipo de la función strcmp() es como sigue:

int strcmp(const char *s1, const char *s2)

Explicación: Su nombre proviene de string comparison. Sirve para comparar dos cadenas de caracteres. Como argumentos utiliza punteros a las cadenas que se van a comparar. La función devuelve cero si las cadenas son iguales, un valor menor que cero si s1 es menor –en orden alfabético– que s2, y un valor mayor que cero si s1 es mayor que s2. La función strcomp() es completamente análoga, con la diferencia de que no hace distinción entre letras mayúsculas y minúsculas).

Función Strcpy()

El prototipo de la función strcpy() es como sigue:

char *strcpy(char *s1, const char *s2)

Explicación: Su nombre proviene de string copy y se utiliza para copiar cadenas. Utiliza como argumentos dos punteros a carácter: el primero es un puntero a la cadena copia, y el segundo es un puntero a la cadena original. El valor de retorno es un puntero a la cadena copia s1. Es muy importante tener en cuenta que en C no se pueden copiar cadenas de caracteres directamente, por medio de una sentencia de asignación. Por ejemplo, sí se puede asignar un texto a una cadena en el momento de la declaración:

```
char s[] = "Esto es una cadena"; // correcto
```

Sin embargo, sería ilícito hacer lo siguiente:

```
char s1[20] = "Esto es una cadena";
char s2[20];
...
// Si se desea que s2 contenga una copia de s1
s2 = s1; // incorrecto: se hace una copia de punteros
strcpy(s2, s1); // correcto: se copia toda la cadena
```

Punteros como valor de retorno

A modo de resumen, recuérdese que una función es un conjunto de instrucciones C que:

- Es llamado por el programa principal o por otra función.
- Recibe datos a través de una lista de argumentos, o a través de variables extern.

función el nombre de otra función. Por ejemplo, si pfunc es un puntero a una función que devuelve un entero y tiene dos argumentos que son punteros, dicha función puede dec lararse del siguiente modo:

int (*pfunc)(void *, void *);

El primer paréntesis es necesario pues la declaración:

```
int *pfunc(void *, void *); // incorrecto
```

corresponde a una función llamada pfunc que devuelve un puntero a entero. Considérese el siguiente ejemplo para llamar de un modo alternativo a las funciones sin() y cos(x):

```
#include <stdio.h>
#include <math.h>
void main(void){
double (*pf)(double);
*pf = sin;
printf("%lf\n", (*pf)(3.141592654/2));
*pf = cos;
printf("%lf\n", (*pf)(3.141592654/2));
}
```

Obsérvese cómo la función definida por medio del puntero tiene la misma "signature" que las funciones seno y coseno. La ventaja está en que por medio del puntero pf las funciones seno y coseno podrían ser pasadas indistintamente como argumento a otra func ión.

Argumentos
Paso De Argumentos Por Valor Y Por Referencia

Anteriormente se ha comentado que en la llamada a una función los ***argumentos actuales*** son evaluados y se pasan ***copias*** de estos valores a las variables que constituyen los ***argumentos formales*** de la función. Aunque los argumentos actuales sean variables y no expresiones, y haya una correspondencia biunívoca entre ambos tipos de argumentos, los cambios que la función realiza en los argumentos formales no se trasmiten a las variables del programa q ue la ha llamado, precisamente porque lo que la función ha recibido son *copias*. El
modificar una copia no repercute en el original. A este mecanismo de paso de argumentos a una función se le llama ***paso por valor***.

Considérese la siguiente función para per mutar el valor de sus dos argumentos x e y:

```
void permutar(double x, double y) // funcion incorrecta
{
double temp;
temp = x;
x = y;
y = temp;
}
```

La función anterior podría ser llamada y comprobada de la siguiente forma:

```
#include <stdio.h>
void main(void)
{
double a=1.0, b=2.0;
void permutar(double, double);
printf("a = %lf, b = %lf\n", a, b);
permutar(a, b);
printf("a = %lf, b = %lf\n", a, b);
}
```

Compilando y ejecutando este programa se ve que a y b siguen teniendo los mismos valores antes y después de la llamada a permutar(), a pesar de que en el interior de la función los valores sí se han permutado (es fácil de comprobar introduciendo en el código de la función los printf() correspondientes). La razón está en que se han permutado los valores de las copi as de a y b, pero no los valores de las propias variables. Las variables podrían ser permutadas si se recibieran sus direcciones (en realidad, copias de dichas direcciones).

Las direcciones deben recibirse en variables puntero, por lo que los argumentos formales de la función deberán ser punteros. Una versión correcta de la función permutar() que pasa direcciones en vez de valores sería como sigue:

```
void permutar(double *x, double *y)
{
double temp;
temp = *x;
*x = *y;
*y = temp;
}
```

que puede ser llamada y comprobada de la siguiente forma:

```
#include <stdio.h>
void main(void)
{
double a=1.0, b=2.0;
void permutar(double *, double *);
printf("a = %lf, b = %lf\n", a, b);
permutar(&a, &b);
printf("a = %lf, b = %lf\n", a, b);
}
```

Al mecanismo de paso de argumento s mediante direcciones en lugar de valores se le llama paso por referencia, y deberá utilizarse siempre que la función deba devolver argumentos modificados. Un caso de particular interés es el paso de arrays (vectores, matrices y cadenas de caracteres). Basta decir ahora que como los nombres de los arrays son punteros (es decir, direcciones), dichos datos se pasan por referencia, lo cual tiene la ventaja adicional de que no se gasta memoria y tiempo para pasar a las funciones copias de cantidades grandes de información.

Un caso distinto es el de las estructuras, y conviene tener cuidado. Por defecto las estructuras se pasan por valor, y pueden representar también grandes cantidades de datos (pueden contener arrays como miembros) de los que se realizan y tr ansmiten copias, con la consiguiente pérdida de eficiencia. Por esta razón, las estructuras se suelen pasar de modo explícito por referencia, por medio de punteros a las mismas.

La función main() con argumentos

Cuando se ejecuta un programa desde MS-DOS tecleando su nombre, existe la posibilidad de pasarle algunos datos, tecleándolos a continuación en la misma línea. Por ejemplo, se le puede pasar algún valor numérico o los nombres de algunos ficheros en los que tiene que leer o escribir información. Esto se consigue por medio de argumentos que se pasan a la función main(), como se hace con otras funciones.

Así pues, a la función main() se le pueden pasar argumentos y también puede tener valor de retorno. El primero de los argumentos de main() se suele l lamar argc, y es una variable int que

contiene el número de palabras que se teclean a continuación del nombre del programa cuando éste se ejecuta. El segundo argumento se llama argv, y es un vector de punteros a carácter que contiene las direcciones de la primera letra o carácter de dichas
palabras. A continuación se presenta un ejemplo:

```
int main(int argc, char *argv[])
{
int cont;
for (cont=0; cont<argc; cont++)
printf("El argumento %d es: %s \n", cont, argv[cont]);
printf("\n");
return 0;
}
```

- Realiza una serie de tareas específicas, entre las que pueden estar cálculos y operaciones de lectura/escritura en el disco, en teclado y pantalla, etc.
- Devuelve resultados al programa o función que la ha llamado por medio del valor de retorno y de los argumentos que hayan sido pasados por referencia (punteros).

El utilizar punteros como valor de retorno permite superar la limitación de devolver un único valor de retorno. Puede devolverse un puntero al primer elemento de un vector o a la dirección base de una matriz, lo que equivale a devolver múltiple valores. El valor de retorno puntero a void (void *) es un puntero de tipo indeterminado que puede asignarse sin casting a un puntero de cualquier tipo. Los punteros a void son utilizados por las funciones de reserva dinámica de memoria calloc() y malloc(), como se verá más adelante.

Ambito de las variables

El ámbito de una variable es el contexto dentro del que la variable está definida. La mayor parte de las variables PHP sólo tienen un ámbito simple. Este ámbito simple también abarca los ficheros incluidos y los requeridos.

Definición de variables

Una variable es un nombre asignado a una posición de almacenamiento de datos. El programa utiliza variables para guardar varios tipos de datos durante la ejecución del programa. En C, una variable debe ser definida antes de que pueda ser usada. Una definición de variable le informa al compilador el nombre de la variable y el tipo de datos que va a guardar. Las variables se pueden declarar en la zona de encabezado de un programa o al inicio de una función o un procedimiento.

Variables globales:	**Variables Locales:**
Las variables globales son aquellas variables que se definen o declaran en la zona de encabezado de cualquier programa en C. Estas variables pueden ser utilizadas en cualquier parte del programa, igualmente puede ser modificado su valor desde cualquier instrucción.	Las variables son consideradas como locales cuando su declaración se hace al inicio de una función o un procedimiento. Las variables que hayan sido declaradas como locales solo podrán ser reconocidas por el procedimiento o función donde se haya declarado. En ninguna otra parte del programa se puede hacer uso de ellas.
Un reciente cambio de regla de aproximación del comité de ANSI de C++ afecta la visibilidad de una variable que se declara en una instrucción como for. El siguiente código generará un error de compilador. Int index; For (int i=1;I<10;I++) { cout <<I;} index=I;	

Definición de variables globales

Una varible global se declara fuera de todas las funciones, incluyendo a la función main(). Una variable global puede ser utilizada en cualquier parte del programa.

Por ejemplo:

```
short numero, suma;
int numerogr, sumagr;
char letra;

main()
{
...
}
```

Es también posible preinicializar variables globales us ando el operador de asignación =, por ejemplo:

```
float suma= 0.0;
int sumagr= 0;
char letra= 'A';

main()
{
...
}
```

Que es lo mismo que:

```
float suma;
int sumagr;
char letra;

main()
{
   suma  = 0.0;
   sumagr= 0;
   letra = 'A';

...
}
```

Dentro de C también se permite la asignación múltiple usando el operador =, por ejemplo:

```
a = b = c = d = 3;
```

...que es lo mismo, pero más eficiente que:

```
a = 3;
b = 3;
c = 3;
d = 3;
```

La asignación múltiple se puede llevar a cabo, si todos los tipos de las variables son iguales .

Se pueden redefinir los tipos de C usando typedef. Como un ejemplo de un simple uso se considera como se crean dos nuevos tipos real y letra. Estos nuevos tipos pueden ser usados de igual forma como los tipos predefinidos de C.

```
typedef float real;
typedef char letra;

/* Declaracion de variables usando el nuevo tipo */
real suma=0.0;
letra sig_letra;
```

Lectura y escritura de variables

El lenguaje C usa salida formateada. La función printf tiene un caracter especial para formatear (%) -- un caracter enseguida define un cierto tipo de formato para una variable.

%c caracteres
%s cadena de aracteres
%d enteros
%f flotantes

Por ejemplo:

```
printf("%c %d %f",ch,i,x);
```

La sentencia de formato se encierra entre " ", y enseguida las variables. Asegurarse que el orden de formateo y los tipos de datos de las variables coincidan.

scanf() es la función para entrar valores a variables. Su formato es similar a printf. Por ejemplo:

```
scanf("%c %d %f %s",&ch, &i, &x, cad);
```

Observar que se antepone & a los nombres de las variables, excepto a la cadena de caracteres.