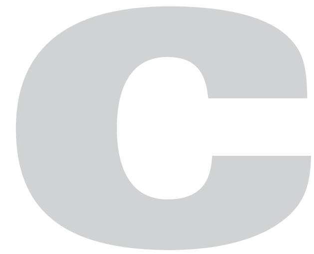
Programación en lenguaje



Referencia Rápida

Jorge Juan Gómez Basanta

Derechos Reservados 1997

Contenido

1. Cabecera del programa: directivas de preprocesamiento (include y define)....4
2. Declaraciones y definiciones:....4
3. Uso de constantes 5
4. Tipos de datos simples 6
Nombres de identificadores 6
Especificadores de tipo de almacenamiento.... 7
5. Programa principal main y funciones 7
La proposición return.... 9
Recursividad.... 10
Notas sobre la sintaxis de C.... 10
6. Uso de parámetros.... 10
7. Introducción a las librerias de funciones.... 11
Funciones de entrada y salida: 12
printf().... 12
scanf.... 13
Cómo utilizar un juego de inspección 13
Otras funciones comunes.... 14
CONIO.H 14
CTYPE.H.... 14
DOS.H 14
MATH.H y COMPLEX.H.... 14
STDIO.H 15
STRING.H.... 15
8. Operadores 15
Operador de asignación..... 15
Operadores aritméticos.... 15
Operadores relacionales y lógicos 16
El operador condición.... 16
9. Sentencias de control de la lógica.... 16
if then else.... 16
switch 18
10. Sentencias iterativas (bucles, ciclos).... 19
for.... 19
while 19
do-while.... 20

11. Tipos de datos complejos 20
Arreglos unidimensionales 20
Arreglos multidimensionales 21
Cadenas (Strings) 22
strcpy (cadena1, cadena2) 22
strcat (cadena1, cadena2) 23
strlen (cadena1); 23
strcmp (cadena1, cadena2) 23
Conversión de cadenas a tipos de datos numéricos y viceversa 23
gcvt (variable_numerica, c_sign, cadena) 23
atof , atoi 23
itoa 24
strtod 24
12. Estructuras y enumeraciones. Uso de typedef 24
Estructuras 24
Enumeraciones 25
typedef 25
13. Pantalla Gráfica 26
Funciones de la librería GRAPHICS.H 26
arc(x, y, ángulo_inicial, ángulo_final, radio); 26
bar(x1, y1, x2, y2); 26
circle(x, y, radio); 26
cleardevice(); 26
closegraph(); 26
detectgraph; 26
drawpoly(numpuntos, puntos_polígono); 27
ellipse(x, y, xradio, yradio); 27
fillellipse(x, y, xradio, yradio); 27
fillpoly(numpuntos, puntos_polígono); 27
floodfill(x, y, color_del_borde); 27
getimage(x1, y1, x2, y2, apuntador); 27
imagesize(x1, y1, x2, y2); 27
initgraph(&gdriver, &gmode, "path_de_BGI"); 27
line(x1, y1, x2, y2); 27
moveto(x, y); 27
outtext(cadena); 27
outtextxy(x, y, cadena); 27
pieslice(x, y, ángulo_inicial, ángulo_final, radio); 28
putimage(x, y, apuntador, tipo_de_colocación); 28
putpixel(x, y, color); 28
rectangle(x1, y1, x2, y2); 28
setbkcolor(color); 28
setcolor(color); 28
setfillpattern(patrón, color); 28
setfillstyle(patrón, color); 28

setlinestyle(estilo_de_línea, patrón, color); 28
settextjustify(justificación); 28
settextstyle(fuente, dirección, tamaño); 28
Movimiento de imágenes. Aplicación de los apuntadores o punteros. 28

Apéndice "A" 32
Fuentes 32
Patrones 32
Colores 32
Ancho de las líneas 33
Estilos de líneas 33
Valores para el último argumento de la instrucción putimage() 33

a) Cabecera del programa: directivas de preprocesamiento (include y define)

b) Declaraciones y definiciones:

 declaración de variables (definiendo variables globales, locales)

 declaración de funciones

c) Uso de constantes

c) Tipos de datos simples: int, char, float, double, void (con los modificadores signed, unsigned, long, short) Nombres de modificadores. Variables.

d) Especificadores de clase de almacenamiento (extern, static)

d) Programa principal (main()) y funciones. La proposición return. Recursividad.

e) Uso de parámetros

d) Instrucciones básicas para escribir y leer valores de la pantalla, limpiar la pantalla, posicionar texto. Introducción a las bibliotecas (libraries)

e) Sentencias de asignación. Operadores aritméticos, relacionales. Operador de condición

e) Sentencias de control de la lógica: if then else, switch case

f) Sentencias iterativas (ciclos): for, while, do while

g) Tipos de datos complejos: Arreglos unidimensionales, arreglos multidimensionales, cadenas (strings)

h) Registros, estructuras, uniones, enumeraciones. Uso de typedef.

i) Uso de pantalla gráfica. Movimiento de imágenes a través de punteros. Direccionamiento de memoria.

j) Archivos: acceso directo y acceso secuencial.

1. Cabecera del programa: directivas de preprocesamiento (include y define)

Todo programa de C inicia con una serie de declaraciones que indican al compilador como transformar el programa a lenguaje de máquina. Al principio de cada programa deben incluirse las directivas de preprocesamiento, que indican al compilador cuales serán las librerias (con terminación .h) y los archivos externos de los que dependerá el programa para funcionar adecuadamente. La directiva #include indica al compilador que ha de incluir un archivo fuente. Es importante que cada directiva esté en una línea propia. El nombre de dicho archivo debe estar entre comillas o entre paréntesis angulares, por ejemplo:

```
#include “stdio.h”
#include <stdio.h>
```

Otra directiva importante es #define, que define un identificador y una secuencia de caracteres que sustituirá al identificador cada vez que este aparezca a lo largo del listado. Por ejemplo,

```
#define VERDADERO 1
#define FALSO 0
```

Cada vez que el compilador encuentre el identificador VERDADERO, lo sustituirá por 1, y cada aparición de FALSO será sustituida por 0. Esto es en cierto modo una manera de definir constantes.

2. Declaraciones y definiciones:

Ahora es necesario declarar las variables globales en caso de existir. En este punto también se definir las estructuras, definiciones de tipos… En el ejemplo que aparece a continuación, se aprecia como declarar una variable entera, int, con la línea `int opción;` . Una variable es una posición de memoria que se utiliza para mantener un valor que puede ser modificado durante el programa. Cabe destacar que la variable que aparece en el ejemplo es global, por lo que es posible utilizarla en cualquier parte del programa sin que pierda su valor. Las variables locales, son aquellas que se definen dentro del cuerpo de alguna función o de main(), y que sólo conservan su valor en aquella parte del programa en que fueron definidas. Las variables globales, sin embargo, tiene la desventaja de ocupar memoria (RAM) durante todo el tiempo de ejecución del programa, en entornos donde la memoria es escasa es importante considerar esto.

Ejemplo:

```
#include <stdio.h>
int cuadrado(int x);
int opcion;
…
void main {…}

int cuadrado(int x)
{ … }
```

A continuación, se deben escribir las cabeceras de todas las funciones (un poco más adelante se explicará con detalle el concepto de función) que aparecen en el programa, seguidas de un punto y coma. La cabecera que se escribe debe ser idéntica a la que aparece en el cuerpo del programa, como en el ejemplo anterior, que se define una función que regresará un valor entero con la línea int cuadrado (int x).

3. Uso de constantes

Las constantes están referidas a valores fijos que no cambian durante la ejecución del programa; pueden ser de cualquier tipo de dato básico. Son generalmente utilizadas para definir números en base octal o hexadecimal, útiles para algunos manejos de datos.
Ejemplo:

```
int hex = 0x80; // el número 128 en decimal
int oct = 012;  // 10 en decimal
```

Una constante hexadecimal comienza con 0x y le sigue el número en hexadecimal. Una constante en octal empieza con O.

Nota: El escribir dos diagonales como en el ejemplo anterior, indica al compilador que debe ignorar lo que aparece después de estas, es decir, se pone "entre comentarios". Si se desea poner una porción mayor del programa entre comentarios, se marca el inicio con /* y se termina con */.

Constantes de carácter.

Algunos caracteres son imposibles de introducir desde el teclado, como por ejemplo un retorno de carro o una comilla (debido a que para imprimir un caracter en la pantalla este debe encerrarse entre comillas). Por este motivo existen ciertas constantes que permiten escribir estos caracteres especiales.

\b	Espacio atrás (backspace)	\n	Salto de línea	\\	Barra invertida
\f	Salto de página	\r	Salto de carro	\v	Tabulador vertical
\"	Comillas dobles	\’	Comillas simples	\a	Alerta
\0	Nulo	\t	Tabulación horizontal		

4. Tipos de datos simples

Existen cinco tipos de datos en C: **char**, **int**, **float**, **double** y **void**. **Char** guarda caracteres dentro del juego de caracteres ASCII; también pueden guardar números de -127 a 127 este tipo de dato ocupa 8 bits en memoria. Los datos **int** ocupan 16 bits y guardan números enteros de -32767 a 32767. Los valores tipo **float** guardan números con punto flotante, con seis dígitos de precisión y ocupan 32 bits. Finalmente, los valores **double** ocupan 64 bits y tienen diez dígitos de precisión. Existen ciertos modificadores que alteran el tipo de valores que guardan estos tipos base. Los modificadores son: **signed**, **short**, **long** y **unsigned**. Al aplicar un modificador signed a un tipo base, este tipo puede guardar números positivos o negativos. Del mismo modo, un modificador unsigned guarda sólo los números, sin signo; así, un valor entero sin signo puede guardar valores desde 0 hasta 65535, y un char de 0 a 255.

El tipo **void**, puede declarar explicitamente que una función no devuelve valor alguno o bien sirve para crear punteros genéricos.

Ejemplos:

```
signed char numero;
signed long int cantidad;
float _distancia;
```

Nombres de identificadores

Todos los objetos definidos por el usuario, como funciones, etiquetas, variables y constantes se conocen como identificadores. El primer carácter de todo identificador puede ser una letra o un símbolo de subrayado. El resto pueden ser números o letras o símbolos de subrayado. Cabe destacar que C es un lenguaje sensible a las mayúsculas, por lo que CONTADOR, contador y Contador son diferentes identificadores.

Especificadores de tipo de almacenamiento

Existen dos especificadores de tipo de almacenamiento básicos: **extern** y **static**. Estos especificadores indican al compilador como almacenar determinada variable, de acuerdo a como se utilice. El especificador **extern** es útil cuando un programa se encuentra en varios archivos diferentes, y existen variables globales que se desean utilizar a lo largo de todo el programa. Supongamos que contamos con un primer archivo llamado Programa1.cpp, que contien la primera parte de nuestro programa y con un segundo archivo Programa2.cpp que tiene las funciones mas utilizadas en nuestro programa. En el programa 1 definiremos las variables globales que utilizaremos en todo el programa. Entonces es necesario definirlas como

```
int base,altura;  // en el archivo Programa1.cpp
```

y hacer referencia de ellas en el segundo archivo como variables externas,

```
extern int base,altura; // en el archivo Programa2.cpp
```

De este modo, no se crea un doble almacenamiento para las variables.

Normalmente, las variables locales a una función sólo mantienen su valor durante la ejecución de dicha función o en caso de ser globales mientras se llame a funciones dentro del mismo archivo. Las variables globales con especificador **static**, no pierden su valor cuando termina la ejecución de determinada función o cuando se llama a una función en otro archivo. Una variable local **static** es una variable local que retiene su valor entre llamadas de funciones.

Cabe destacar que al momento de definir una variable, esta adquiere un valor "basura". Por este motivo es muy conveniente inicializar todas las variables en algún valor antes de utilizarlas, para evitar errores.

5. Programa principal main y funciones

Un programa en C, esta constituido básicamente por declaraciones, definiciones y funciones. Existe una función principal que es la primera en ser llamada al momento de ejecutar un programa. Esta función se conoce como **main()** De hecho, todo el programa puede residir

dentro de esta función, pero la mayoría de los programas se tornarían redundantes y poco eficientes al hacer esto. A continuación aparece un ejemplo de un programa muy simple:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
      printf("Hola mundo!");
      delay(5000);
      return 0;
}
```

El programa anterior escribe "Hola mundo!" en la pantalla. Por conveniencia, se acostumbra dividir un programa complejo en varias partes, para así poder aislar una de ellas en caso de causar problemas y para hacer más fácil el mantenimiento de dicho programa en caso de necesitar modificaciones. A este tipo de programación basado en la creación de estructuras para el control del programa se le conoce como programación estructurada.

Una función se define del modo siguiente.

tipo_de_dato identificador (listado_de_parámetros)

```
#include <stdio.h>
#include <conio.h>
int cuadrado(int num);

void main(void)
{
      int numero = 0;
      int resultado = 0;
      printf("Dame una cantidad: ");
      scanf("%d",&numero);
      resultado = cuadrado(numero);
      printf("El resultado es %d : ",resultado);
      delay(5000);
      return 0;
}

int cuadrado(int num)
{
      return num * num;
}
```

Este programa trabaja del siguiente modo. Primero se pide una cantidad al usuario con la instrucción printf, que escribe texto en la pantalla. El valor que proporciona el usuario es guardado en la variable numero, con la instrucción scanf (que utiliza un argumento "%d" debido a que se desea guardar un entero). Después, la variable resultado guarda el valor devuelto por la función, al llamar a la función por su identificador con un parámetro, con la línea resultado = cuadrado(numero); .

La función cuadrado tiene una proposición return, que indica indica el valor que debe devolver cuando sea llamada cabe destacar que el valor que devuelve la función debe ser del mismo tipo con el cual se definió. Es decir, la función definida por

```
int cuadrado(int num)
```

debe devolver un valor entero. Así, una función definida como

```
char menu(void)
```

debe devolver un caracter (char). Cabe destacar que esta última función no recibe parámetros, como lo indica el tipo de dato void entre paréntesis después del identificador de la función.

Cuando termina la ejecución de la función cuadrado, se devuelve el control al programa principal (main), y se escribe la variable resultado utilizando la instrucción printf.

No todas las funciones deben regresar algún valor. Muy comúnmente, se utilizan funciones para dividir un programa en segmentos. En este tipo de funciones se debe indicar que no regresa valores, con un tipo de dato void, en estos casos, no es necesario escribir la proposición return ya que devuelven valores nulos.

Ejemplo:

```
void saludo(void)
{
      printf("Saludos a todos los programadores de C";
}
```

Una función es llamada al escribir el nombre de la función y los parámetros de esta entre paréntesis (en caso de existir). Así, si quisieramos llamar a la función saludo desde main, escribiriamos los siguiente,

```
void main()
{
      saludo(); // sólo se abren y cierran paréntesis ya que esta función no recibe
parámetros.
}
```

La proposición return

La proposición return sirve para indicar que valor va a devolver una función. Simultáneamente, sirve para salir de la función. Es decir, al regresar un valor, la función regresa el control a la función desde donde fue llamada, aunque existan más líneas después de la proposición return. Es frecuente crear funciones que devuelvan valores tipo int que convencionalmente devuelvan un valor 1 o 0, para indicar si el resultado de la función fue verdadero o falso respectivamente.

Por otro lado, la función main() devuelve un entero al proceso de llamada, que generalment es el sistema operativo. Si main no devuelve un valor explicitamente, el valor queda indefinido. Si main() no devuelve un valor debe declararse de tipo void o en su defecto simplemente poner return 0; antes de la llave que cierra a la función.

Recursividad

En C, las funciones pueden llamarse a sí mismas. Si una expresión en el cuerpo de una función llama a la propia función, se dice que esta es **recursiva.** Suele llamarse a veces definición circular. Un ejemplo de función recursiva tradicional es la utilizada para obtener el factorial:

```
int factorial(int n)
{
      int resultado;
      if (n == 1) return 1;
      resultado = factorial(n-1) * n;
      return (resultado);
}
```

Notas sobre la sintaxis de C

Cabe destacar algunas detalles de la sintaxis de C. Como se habrá notado, el cuerpo de una función siempre se encierra entre llaves. Lo mismo sucede en el caso de sentencias condicionales y de iteración. Todo lo que se quiera incluir dentro de una condición o un ciclo debe encerrarse entre llaves salvo en el caso de que se trate de una sola línea (como en el ejemplo anterior en el caso del if que aparece en la cuarta línea). Todas las sentencias compuestas llevan un punto y coma al final. Las sentencias de iteración y condicionales nunca llevan punto y coma. Por otro lado, todas las llamadas a funciones deben estar formadas por el nombre de la función y entre paréntesis el o los parámetros si es que existen. Cuando no existen parámetros, simplemente se abren y cierran paréntesis. Una costumbre muy buena es la de indentar los programas, de este modo se pueden identificar facilmente las estructuras.

6. Uso de parámetros

Hasta ahora, se ha mencionado la palabra parámetro en varias ocasiones pero no ha sido definido su concepto. Un parámetro es simplemente un valor que recibe una función cuando esta es llamada con argumentos. Es necesario indicar de que tipo es el valor que se recibe y

este debe ser el mismo del valor enviado. Los parámetros son almacenados en variables que se comportan como cualquier otra variable local. El siguiente ejemplo ilustra el concepto de parámetro.

```
#include <stdio.h>
#include <conio.h>
void escribeCoordenadas(int x, int y)

void main(void)
{
      int coordx,coordy;
      int resultado = 0;
      printf("Dame la coordenada en x: ");
      scanf("%d",&coordx);
      printf("Dame la coordenada en y: ");
      scanf("%d",&coordy);
      escribeCoordenadas(coordx,coordy);
      delay(5000);
      return 0;
}

void escribeCoordenadas(int x, int y)
{
      printf("La coordenada en x es: %d",x);
      printf("La coordenada en y es: %d",y);
}
```

En este programa se envian desde la función principal (main()) los valores leidos de la pantalla a través de scanf, con la sentencia escribeCoordenadas(coordx,coordy). La función recibe los valores en las variables x y y de tipo entero, y las escribe en la pantalla.

7. Introducción a las librerias de funciones

A continuación aparece una lista de funciones comunes contenidas en librerias que son utilizadas frecuentemente. Aunque aquí aparecen las funciones básicas, existen muchas más que quedan por investigar para el programador de C. Algunas librerias comunes son: conio.h, dos.h, stdio.h, stdlib.h, string.h, math.h, graphics.h y ctype.h. Algunas de las funciones más comunes de cada una de estas librerías serán presentadas posteriormente.

Funciones de entrada y salida:

printf()

Se escribe de la siguiente manera:

```
printf(cadena_de_control).
```

Esta cadena de control tiene el tipo char * (apuntador o puntero). Esta constituida por dos tipos de elementos. El primero formado por caracteres que se mostraran en pantalla y el segundo formado por especificadores de formato, que indicarán como escribir variables en la pantalla. Un especificador de formato empieza con un signo de porcentaje seguido por una literal que representa al formato. Debe haber exactamente el mismo número de argumentos como especificadores de formato. Por ejemplo,

```
printf("Los resultados son: %d , %d %c", numero1, numero2, '.');
```

donde numero1 y numero2 son variables enteras.

Lista de especificadores de formato:

%c	carácter
%d	Enteros con signo
%e	Notación científica
%f	Número con punto flotante
%o	Octal sin signo
%s	Cadena de caracteres
%u	Enteros sin signo
%x	Hexadecimales sin signo
%X	Hexadecimales con signo
%%	Escribe el signo %

Para indicar la precisión con la que se debe escribir una cantidad de punto flotante o a una cadena de caracteres. Por ejemplo, si se escribe %5.3f, se indica que se desean cinco enteros y tres cifras decimales. Si se aplica a cadenas, el especificador de precisión indica la longitud máxima del campo. Por ejemplo, al escribir %6.9s se imprime una cadena con al menos 6 caracteres y no más de 9.

Ejemplo:

```
printf("%3.2f\n", 823.12354);
printf("%10.15s\n", "Esta es una prueba de texto".);
```

Estas líneas desplegarán en pantalla

```
823.12
Esta es una pru
```

scanf

Esta instrucción sirve para guardar en variables los datos proporcionados por el usuario a través del teclado. La sintaxis es la siguiente:

```
scanf(cadena_de_control)
```

La cadena de control está formada por especificadores de formato y nombres de variables,

Los especificadores de formato van precedidos por el signo %, e indican que tipo de dato será leido a continuación.

Especificadores de formato de scanf

%c	Lectura de un carácter
%d	Lectura de un entero
%f	Lectura de un número con punto flotante
%o	Lectura de números octales
%s	Lectura de una cadena
%x	Lectura de números hexadecimales
%[ ]	Muestreo de un conjunto de caracteres

Para leer varios valores al mismo tiempo, se puede utilizar scanf de la siguiente manera

```
scanf("%c%c%c",&a,&b,&c);
```

En este ejemplo se leen tres caracteres juntos en tres variables diferentes.

Si se desean leer valores separados por comas, se puede escribir de este modo:

```
scanf("%d,%d",&num1,&num2);
```

Cabe destacar que el & que aparece antes de las variables sirve para apuntar a la dirección de memoria de dicha variable.

Cómo utilizar un juego de inspección

Un juego de inspección define un conjunto de caracteres. Se define poniendo los caracteres que se pueden leer entre corchetes, precedidos por un signo de %. De este modo, se puede especificar que valores serán leídos por las variables. El siguiente ejemplo ilustra el uso de un juego de inspección.

```
char cadena[80];
scanf("[bcdfghjklmnpqrstvwxyz]%s", cadena);
```

En el ejemplo anterior, sólo se guardan en la cadena las consonantes en minúsculas. Al encontrar la primera vocal, scanf deja de asignar caracteres a cadena. También es posible especificar un rango empleando un guión, como en [A-Z], con lo cual se guardarían únicamente las letras mayúsculas, ya que es sensible a las minúsculas.

Otras funciones comunes

CONIO.H

•clrscr(); Sirve para limpiar la pantalla
•cputs(); Escribe solo texto en la pantalla
•getche(); Lee un valor en una variable sin necesidad de presionar ENTER
•clreol(); Limpia la línea en la que se encuentra el cursor
•textcolor(); Define el color para el texto
•textbackground() Define el color para el fondo
•textmode() Cambia entre el modo de 40 y de 80 columnas
•kbhit() Devuelve verdadero cuando se presiona una tecla
•window() Pinta un cuadro en la pantalla para escribir texto
•gotoxy() Mueve el cursor a una posición en la pantalla (de 80 cols y 25 rens)

CTYPE.H

•toupper() Convierte un caracter a su mayúscula correspondiente
•tolower() Convierte un caracter a su minúscula correspondiente

DOS.H

•delay() Espera un determinado tiempo

MATH.H y COMPLEX.H

•sqrt() Obtiene la raiz cuadrada de un número
•pow() Eleva un número a determinado exponente
•sin() Obtiene el seno de un número
Existen muchas otras funciones de esta librería, para todas las funciones trigonométricas, logaritmos…

STDIO.H

printf( ) y scanf( )

STRING.H

•strcat() Sirve para concatenar o unir dos strings o cadenas
•strcopy() Copia un string o cadena en una variable
•strcmp() Compara dos strings o cadenas
•strlen() Obtiene la longitud de un string

8. Operadores

Operador de asignación.

El operador más común, es el que sirve para dar valor a las variables de manera directa. Funciona igualando una variable a una expresión que puede ser la llamada a una función, una operación matemática o un valor directamente. Ejemplos:

```
int resultado=(sqrt(85)*32)+54;
char opcion='a';
float numero = cuadrado(5);
```

Operador	Acción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo o residuo de una división entera
++	Incremento
--	Decremento
+=	Acumula un valor en una variable. Al escibir por ejemplo: num+=5; essimilara: num=num+5
-=, *=, /=	funcionan de manera similar a +=

Operadores aritméticos

Operador	Acción
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual
!=	No igual
&&	Y (And)
||	O (Or)
!	No (Not)

Operadores relacionales y lógicos

El operador condición

El operador condición ? sustituye a la sentencia if - then - else, que se verá más adelante. Su sintaxis es:

expresion1 ? expresion2 : expresion3

Si expresion1 es verdadera, entonces se ejecuta la expresion2. En caso contrario, se ejecuta la expresión3. Ejemplo:

```
int num = 50;
y = num > 20 ? 500 : 300;
```

En el ejemplo anterior, y toma el valor de 500, ya que 50 es mayor que 20. Se puede sustituir por una sentencia if then else.

9. Sentencias de control de la lógica

if then else

El ejemplo anterior puede ser sustituido por:

```
int num = 50;
if (num > 20)
      y = 500;
else
      y = 300;
```

De este modo se ejemplifica el funcionamiento de la sentencia if then else. Su funcionamiento es simple, si la condición resulta verdadera al evaluarse, entonces se ejecutarán las instrucciones que estén debajo del if. En caso de resultar falsa, el programa salta hasta encontrar un else si es que este existe. En caso de no existir un else, se continuará con el programa, después de saltar las llaves que encierran las instrucciones del if. He aquí un ejemplo que ilustra claramente la sintaxis y el comportamiento de un if. Es importante destacar que si existe más de una condición en un if unida por un operador && (AND) o || (OR), debe haber un paréntesis que encierre a toda la condición.

```
int x,y,z;
printf("Escribe tres números separados por comas");
scanf("%d,%d,%d",x,y,z);
if ((x > y)&&(x>z))
      {
      gotoxy(20,5);
      printf("El primer número es mayor que el segundo y que el tercero.");
      }
else
      {
      if ((x<y)&&(x<z))
            {
            gotoxy(20,5);
            printf("El primer número es menor que el segundo y que el tercero");
            }
      }
```

Otro ejemplo, puede ser un menú de opciones.

```
char opcion;
int x,y,suma,multiplicacion;
printf("Selecciona alguna opcion:");
printf("1. Sumar\n2. Multiplicar");
scanf("%c",opcion);
printf("Escribe dos números separados por comas");
scanf("%d,%d",x,y);
if (opcion == 1)
      {
      suma = x + y;
      printf("La suma es %d",suma);
      }
if (opcion == 2)
      {
      multiplicacion = x * y;
      printf("La multiplicación es %d",multiplicacion);
      }
```

Si se tienen varias sentencias if similares, estas pueden ser sustituidas por un switch case, como se verá a continuación.

switch

Su sintaxis es:

```
switch (expresión)
       {
       case constante1:  {
                             … sentencias …
                             break;
                             }
       case constante2:  {
                             … sentencias …
                             break;
                             }
       default:              {
                             … sentencias …
                             }
       }
```

Para explicarsu funcionamiento se utilizará un ejemplo que hace lo mismo que el último ejemplo de la sentencia if.

```
char opcion;
int x,y,suma,multiplicacion;
printf("Selecciona alguna opcion:");
printf("1. Sumar\n2. Multiplicar");
opcion = getche(); // esta es otra forma de leer un valor del teclado
printf("Escribe dos números separados por comas");
scanf("%d,%d",x,y);

switch (opcion)
       {
       case 1:      {
                    suma = x + y;
                    printf("La suma es %d",suma);
                    break;
                    }
       case 2:      {
                    multiplicacion = x * y;
                    printf("La multiplicación es %d",multiplicacion);
                          break;
                    }
       default:     printf("La opcion seleccionada no es válida");
       }
```

La instrucción break sirve para salir del switch. Cuando ya se cumplió alguno de los casos, no es necesario evaluar los demás casos, entonces se utiliza break. Si el usuario introduce un valor que no está contemplado en el switch, se ejecutan las sentencias de default.

10. Sentencias iterativas (bucles, ciclos)

Este tipo de sentencias sirven para repetir algunas líneas del código.

for

La sintaxis de esta instrucción es la siguiente

```
for (inicialización_de_variables ; condicion ; incremento_o_decremento)
	{
	sentencias a repetir
	}
```

Un ejemplo que explica su funcionamiento puede presentar la raiz cuadrada de los primeros 80 números.

```
int i;
float raiz;
for (i = 0; i <= 80; i++)
	{
	raiz = sqrt(i);
	printf("El cuadrado de %d es %f",i,raiz);
	}
```

El incremento i++ es similar a escribir i = i + 1;. También se puede escribir como ++i.

También se puede escribir un for más complejo, como por ejemplo:

```
for (x=0, y=0 ;  x * y > 30;  x++, j--)
```

Otra alternativa, es un for infinito, que se escribe como se indica a continuación. Para salir de este for es necesario utilizar la sentencia break.

```
for ( ; ;)
```

while

La sintaxis del while es la siguiente:

```
while (condicion)
	{
	sentencias
	}
```

Su funcionamiento es relativamente simple y no requiere de mayor explicación. Mientras la condición arroje un valor verdadero, es decir, se cumpla, se repetiran las sentencias encerradas en el while. Ejemplo:

```
char opcion;
int x,y,suma,multiplicacion;

while ((opcion != 3))
      {
      clrscr();
      printf("Selecciona alguna opcion:");
      printf("1. Sumar\n2. Multiplicar\n3. Salir");
      opcion = getche(); // esta es otra forma de leer un valor del teclado
      }
```

Este while repite el programa hasta que el usuario seleccione la opción 3, que es Salir.

do-while

A diferencia del while y del for, que evaluan una condición antes de ejecutar algún código, el do-while ejecuta por lo menos una vez las instrucciones y luego evalúa la condición.
La sintaxis es:

```
do
      {
      sentencias
      } while (condicion);
```

Ejemplo:

```
int num;
do {
      printf("Dame un número entre 0 y 200: ");
      scanf("%d",num);
      } while ((num >= 0)&&(num <= 200))
```

Este do-while pide un número hasta que este se encuentre en el rango establecido.

11. Tipos de datos complejos

Con base en los cinco tipos de datos básicos, existen datos complejos que permiten controlar mejor el almacen de datos.

Arreglos unidimensionales

Un arreglo es una colección o grupo de variables bajo un mismo nombre. Esta formado por una serie de casillas numeradas que contienen el mismo tipo de datos. Para utilizar un valor guardado en un arreglo es necesario proporcionar el número de la casilla en que está guardado dicho valor. Un arreglo se define de la siguiente manera:

```
tipo_de_dato identificador[número_de_casillas];
```

Un arreglo de enteros, por ejemplo, se puede definir como sigue:
int numeros[100];

La numeración de las casillas de un arreglo empieza en 0, aunque, para fines prácticos, se suele empezar a guardar valores en la casilla número 1. Una de las casillas del arreglo nunca se podrá utilizar (la última), debido a que en esta casilla se guarda el valor que representa el tamaño del arreglo. Para accesar al valor de una casilla, se escribe el nombre del arreglo y entre corchetes el número de la casilla. Así, para hacer referencia al valor 5 del arreglo numeros, se escribiría numeros[5].

Un ejemplo sencillo consiste en preguntar al usuario cuantos datos quiere utilizar, pedirle que escriba dichos datos y proporcionarle el promedio.

```
int i, numero_de_datos, suma = 0, datos[101];
float promedio;
do {
      printf("¿Cuántos datos quieres utilizar? (Máximo 100)");
      scanf("%d",numero_de_datos);
      } while ((numero_de_datos > 1)&&(numero_de_datos < 101)

for (i=0; i <= numero_de_datos; ++i)
      {
      printf("Dame el dato %d \n", i);
      scanf("%d",datos[i]);
      suma+=datos[i];   // suma todos los datos guardándolos en suma
      }

promedio=suma / numero_de_datos;
printf("El promedio de los datos es: %5.2f",promedio);
```

Arreglos multidimensionales

En los arreglos multidimensionales es posible guardar datos de manera similar a una hoja de cálculo. Así, por ejemplo, es posible hacer operaciones entre matrices o definir un sistema de coordenadas.
Los arreglos multidimensionales se definen de la siguiente forma,
tipo_de_dato identificador [tamaño1][tamaño2][tamaño3]…

Así, una matriz de 3 renglones y 2 columnas se definiría como:
int matriz[3][2];

Para acceder al elemento en el renglón 1 y la columna 3, se utilizaría matriz[1][2].
Un ejemplo puede ser aquel que sume todos los elementos de una matriz.

```
int x,y,suma=0, matriz[5][3];        // la variable suma se inicializa en cero
                                     // para que no tome un valor basura
for (x=0; x<5; x++)
      for(y=0; y<3; y++)
            suma+=matriz[x][y];
```

Este arreglo en particular es bidimensional. Para manejar este tipo de arreglos, generalmente se utilizan dos ciclos for anidadas, como en el ejemplo anterior. es importante notar que como cada uno de los for tiene sólo una línea, no se escribieron llaves. También es importante destacar que los valores estan guardados a partir de la posición 0,0 por lo que el último elemento del arreglo se encuentra en matriz[4][2], y NO en matriz[5][3].
Es complicado manejar arreglos de más de 3 dimensiones ya que escapan a nuestra imaginación de un mundo en tres dimensiones.

Cadenas (Strings)

Un caso particular de los arreglos de una dimensión, son las cadenas o strings. En C, una cadena de caracteres es un arreglo que termina con un carácter nulo, que se denota con \0. Por esta razón, el arreglo debe ser definido con un carácter más de lo necesario. Es decir, si se quiere guardar una cadena de 10 caracteres, es necesario definirla con 11.

Para definir una cadena, se escribe

```
char cadena[21];
```

Tal y como se mencionó en el tema 7 (Introducción a las librerías de funciones), existe una librería que sirve para manipular cadenas, llamada STRING.H. A continuación se mostrarán algunas de las funciones de esta librería.

strcpy (cadena1, cadena2)

Copia la cadena1 en la cadena2.

strcat (cadena1, cadena2)

Pone la cadena2 al final de la cadena1.

Ejemplo:

```
char destino[25];
char *cadena_esta= "Esta", *cadena_es="es", *cadena_prueba="una prueba";
strcpy(destino, cadena_esta);
strcat(destino, cadena_es);
strcat(destino, cadena_prueba);
printf("%s\n", destino);
```

strlen (cadena1);

Devuelve un entero igual a la longitud de cadena1, sin contar el carácter terminal.

strcmp (cadena1, cadena2)

Compara la cadena1 con la cadena2. Si son iguales, devuelve un entero mayor que 0. Si la cadena1 es menor que la cadena2, devuelve un entero menor que 0. Devuelve 0 cuando las cadenas son iguales.

Conversión de cadenas a tipos de datos numéricos y viceversa

A veces es necesario poner una variable de tipo int, float o double dentro de una cadena o cambiar una cadena a un formato numérico. Esto se logra mediante instrucciones especiales, que veremos a continuación.

gcvt (variable_numerica, c_sign, cadena)

Sirve para convertir una variable con punto flotante a una cadena. c_sign representa el número de cifras significativas, y debe ser un entero. Ejemplo:

```
char str[25];
double num;
int sig = 5; // número de cifras significativas
num = 9.876;
gcvt(num, sig, str);
printf("string = %s\n", str);
```

atof , atoi

Convierten cadenas de caracteres a variables de tipo float e int respectivamente.
Ejemplos:

```
float f;
char *cad= "12345.67";
f = atof(cad);
printf("cadena= %s número de punto flotante= %f\n", cad, f);


int n;
char *cad= "12345.67";
n = atoi(cad);
printf("cadena= %s entero= %d\n", cad, n);
```

itoa

Covierte una variable tipo int a una cadena. El tercer argumento se refiere a la base en la que está el número que se va a convertir.

Por ejemplo:

```
int numero=53;
char cadena[20];
itoa(numero,cadena,10)
```

strtod

Convierte una cadena a double. Ejemplo:

```
char entrada[80], *punterofin;
double value;
printf("Escribe un número de punto flotante:");
gets(entrada);
valor= strtod(entrada, &punterofin);
printf("Esta cadena es %s . El número es %lf\n", entrada, valor);
```

12. Estructuras y enumeraciones. Uso de typedef

Estructuras

Una estructura es una colección de variables de la que se hace referencia con un mismo nombre. Las variables que componen la estructura se llaman miembros de la estructura.

El formato general para declarar una estructura, es el siguiente.

```
struct identificador {
      tipo_de_dato nombre;
      tipo_de_dato nombre;
      …
} variables de la estructura;
```

Se pueden declarar al instante las variables de estructura, que son variables con ese formato.

Ejemplo:

```
struct fecha {
      int dia;
      int mes;
      int ano;
} fecha_nacimiento, fecha_muerte;

printf("Dame la fecha de nacimiento: dia,mes,año (separados por comas)");
scanf("%d,%d,%d", fecha_nacimiento.dia, fecha_nacimiento.mes, fecha_nacimiento.ano);
```

Enumeraciones

Una enumeración es un conjunto de constantes que especifica todos los valores válidos que una variable de ese tipo puede tener. Las enumeraciones se declaran así:

enum identificador {enumeración} variables;

Ejemplo:

```
enum estado {Sonora, Chihuahua, Guerrero, Tabasco, Yucatán, Veracruz, Puebla,
Morelos, Monterrey, Edo_de_Mex, Queretaro};
```

Después de declarar una enumeración de estados, se puede utilizar de esta manera.

```
if (estado==Sonora) printf("El clima de Sonora es frío");
```

El truco de las enumeraciones es entender que cada elemento es considerado como un númeo entero, empezando desde 0. Así, utilizando la enumeracion definida previamente, se puede escribir,

```
printf("%d",Guerrero);
```

Se escribirá 2 en la pantalla.

Frecuentemente se utilizan las enumeraciones para dar valor a los arreglos. Por ejemplo,

```
char *mes[ ]={"Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
"Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"}
```

Nota: Cuando un arreglo se define sin el tamaño entre corchetes, se trata de un arreglo no delimitado.

typedef

Para definir un nuevo tipo de variable, se utiliza typedef. Esto es, en lugar de utilizar uno de los cinco tipos de datos básicos se pueden crear otros tipos de datos de acuerdo a las necesidades del programa, aunque siempre tomando como base los tipos fundamentales.
Es util definir nuevos tipos para asegurar la portabilidad en varios sistemas operativos, en los que quiza los tipos de datos fundamentales tengan variantes. Al definir nuevo tipos, sólo sera necesario modificar esas líneas para asegurar que el programa funcione. La definiciónde un nuevo tipo es como sigue.

```
typedef tipo nombre;
```

Por ejemplo, para definir un tipo de dato que guarde números imaginarios:

```
typedef struct { double real, imaginaria;  } complejo;
```

13. Pantalla Gráfica

La pantalla gráfica permite crear otro tipo de aplicaciones, al abrir la posibilidad de dibujar varias figuras geométricas, graficar funciones, animar dibujos… Todas las instrucciones de la pantalla gráfica dependen de GRAPHICS.H
Para inicializar la pantalla gráfica, es decir, cambiar del modo de consola o texto al modo gráfico, se puede utilizar las siguiente líneas.

```
int gdriver = DETECT, gmode;
initgraph(&gdriver, &gmode, "C:\\TC\\BGI");
line(0, 0, getmaxx(), getmaxy());
closegraph();
```

Funciones de la librería GRAPHICS.H

La mayoría de estas funciones tienen argumentos similares. He aquí una explicación breve. x, y son variables enteras. Cuando los argumentos son: x1, y1, x2, y2 estos representan a un rectángulo, cuya esquina superior izquierda está en las coordenadas x1, y1 y las coordenadas de la esquina inferior derecha son x2,y2. Todos son valores enteros.

arc(x, y, ángulo_inicial, ángulo_final, radio);

Dibuja un círculo alrededor de las coordenadas x y y con un radio definido por el quinto argumento. El círculo se empieza a dibujar en el sentido de las agujas del reloj a partir de ángulo_inicial y termina en ángulo_final.

bar(x1, y1, x2, y2);

Dibuja un área rectangular rellena de acuerdo con el color definido previamente.

circle(x, y, radio);

Dibuja un círculo con un determinado radio con centro en x,y.

cleardevice();

Limpia la pantalla gráfica

closegraph();

Cierra la pantalla gráfica y libera la memoria apropiada.

detectgraph;

Detecta las variables necesarias para inicializar la pantalla gráfica con initgraph().

drawpoly(numpuntos, puntos_polígono);

Dibuja un polígono de numpuntos tomando las coordenadas de el arreglo de enteros puntos_polígono.

ellipse(x, y, xradio, yradio);

Dibuja una elipse con centro en x, y y con los xradio como radio en x y yradio como radio en y

fillellipse(x, y, xradio, yradio);

Dibuja una elipse rellena del color definido con setcolor() con centro en x, y y con los xradio como radio en x y yradio como radio en y

fillpoly(numpuntos, puntos_polígono);

Dibuja un polígono relleno de numpuntos tomando las coordenadas de el arreglo de enteros puntos_polígono.

floodfill(x, y, color_del_borde);

Rellena un área delimitada por un borde de el color especificado, con el color definido por setcolor().

getimage(x1, y1, x2, y2, apuntador);

Guarda el la imágen delimitada por el rectángulo x1, y1, x2, y2 en la dirección de memoria definida por el apuntador.

imagesize(x1, y1, x2, y2);

Obtiene el número de bytes que una imágen ubicada en x1, y1, x2, y2 ocupa.

initgraph(&gdriver, &gmode, "path_de_BGI");

Sirve para inicializar el modo gráfico.

line(x1, y1, x2, y2);

Dibuja una línea del punto x1, y1 al punto x2, y2.

moveto(x, y);

Sitúa la posición actual en el pixel x, y.

outtext(cadena);

Escribe una cadena en la pantalla gráfica. Para escribir variables en la pantalla, es necesario convertirlas a cadenas con las funciones vistas en el tema 11 (Tipos de datos complejos)

outtextxy(x, y, cadena);

Escribe una cadena en la posición x, y de la pantalla gráfica. Para escribir variables en la pantalla, es necesario convertirlas a cadenas con las funciones vistas en el tema 11 (Tipos de datos complejos)

pieslice(x, y, ángulo_inicial, ángulo_final, radio);

Dibuja un trozo de diagrama de tarta con centro en x, y, con un determinado radio y que comienza en ángulo_inicial y termian en ángulo_final.

putimage(x, y, apuntador, tipo_de_colocación);

Coloca una imagen guardada en un apuntador (con la instrucción getimage()) en la posición x, y de acuerdo con el tipo_de_colocación. Los valores para este último argumento se encuentran en el apéndice A.

putpixel(x, y, color);

Dibuja un sólo pixel de determinado color en la posición x, y de la pantalla

rectangle(x1, y1, x2, y2);

Dibuja un rectángulo.

sector(x, y, ángulo_inicial, ángulo_final, radioX, radioY);

Dibuja un sector (parte de una elipse) con centro en x, y, con un radio horizontal radioX y un radio vertical radioY, que comienza en ángulo_inicial y termian en ángulo_final.

setbkcolor(color);

Establece el color de fondo. (Sólo se pueden usar los primeros 7 colores.)

setcolor(color);

Establece el color para figuras, texto, …

setfillpattern(patrón, color);

Establece el patrón utilizado para rellenar áreas con las instrucciones fillpoly y floodfill.

setfillstyle(patrón, color);

Establece el patrón usado para rellenar áreas de gráficos.

setlinestyle(estilo_de_línea, patrón, color);

Establece el estilo, patrón y grosor de las líneas.

settextjustify(justificación);

Establece como se justificará el texto, justificación puede ser HorizDir o VertDir

settextstyle(fuente, dirección, tamaño);

Establece la fuente, la dirección y el tamaño de las letras desplegadas con outtextxy(). El Apéndice A contiene una tabla con dichos valores.

Movimiento de imágenes. Aplicación de los apuntadores o punteros.

A continuación se ilustra con un ejemplo una de las aplicaciones más interesantes de la pantalla gráfica: el movimiento de una imagen. En este caso, se moverá un polígono, en una trayectoria circular.

```
#include<conio.h>
#include<graphics.h>
#include<dos.h>
#include<stdlib.h>
#include<alloc.h>
#include<math.h>
void dibtriangulo(void);
void preparamovim(void);

//***********************dibtriangulo*****************************************)
void dibtriangulo(void)
{
  int poly[6]={3,3,3,20,30,20}; //se define un arreglo con los puntos del triangulo
  setlinestyle(0,0,1);
  setcolor(4);
  setfillstyle(1,4);
  fillpoly(3,poly);
}

//**********************preparamovim**********************************)
// asigna la memoria necesaria para un triangulo, y la guarda en un apuntador
// *triangle
void preparamovim(void)
{
      void *triangle;
      unsigned int tamano;
```

```
	dibtriangulo();
	tamano = imagesize(0,0,33,23);      //se obtiene el tamano de la imagen
	triangle = malloc(tamano);        //se reserva el espacio en memoria para la
imagen

	getimage(0,0,32,22,triangle);  //se copia la imagen al apuntador
	int y=0;
	cleardevice();
	for(int x=0;x<=400;x=x+2)
	{

	putimage(x,y, triangle, 1);
	delay(2);
	y=(  +sqrt(40000-pow(x-200,2))  )+200;
	putimage(x,y, triangle, 1);
	}

	for(x=400;x>=0;x=x-2)   // el for controla la coordenada en x del triangulo
	{

	putimage(x,y, triangle,1);
	                  // Escribe la imagen en la pantalla
	delay(2);
	y=(  -sqrt(40000-pow(x-200,2))  )+200;
	                  // Obtiene la coordenada en y para el movimiento circular
	putimage(x,y, triangle,1);
	                  // Borra la imagen de la pantalla
	}
	free(triangle); //libera el espacio ocupado por el apuntador triangle
}
//********************PRINCIPAL****************************************)
main()
{
  int gdriver = DETECT, gmode;
  initgraph(&gdriver, &gmode,"C:\\TC\\BGI");
  cleardevice();
  dibtriangulo();
  preparamovim();
  closegraph();
  return 0;
}
```

Bibliografía

MATTSON, Jeff, BORENSTEIN, Philip, et al, Think C, C Development Environment, California, E.U.A., 1993.
832 págs.

SHILDT, Herbert, C++, The Complete Reference, 2a. edición, McGraw-Hill Inc, Estados Unidos, 1995.
592 págs.

Apéndice "A"

Fuentes

DEFAULT_FONT	0	Fuente monoespaciada
TRIPLEX_FONT	1	Fuente estilizada
SMALL_FONT	2	Fuente pequeña
SANS_SERIF_FONT	3	Fuente sans-serif
GOTHIC_FONT	4	Fuente gótica

Patrones

EMPTY_FILL	0	Color de fondo
SOLID_FILL	1	Relleno sólido
LINE_FILL	2	—
LTSLASH_FILL	3	///
SLASH_FILL	4	///, líneas delgadas
BKSLASH_FILL	5	\\\, líneas gruesas
LTBKSLASH_FILL	6	\\\
HATCH_FILL	7	Cuadros inclinados
XHATCH_FILL	8	Cuadros inclinados denso
INTERLEAVE_FILL	9	Líneas intercaladas
WIDE_DOT_FILL	10	Puntos distanciados
CLOSE_DOT_FILL	11	Puntos muy unidos
USER_FILL	12	Patrón definido por el usuario

Colores

(Nota, los colores del 7 al 15 no pueden ser utilizados en la instrucción setbkcolor(), ni en textbackground())

BLACK	0	negro
BLUE	1	azul
GREEN	2	verde
CYAN	3	cian (verde azuloso)
RED	4	rojo
MAGENTA	5	magenta

BROWN	6	café
LIGHTGRAY	7	gris claro
DARKGRAY	8	gris oscuro
LIGHTBLUE	9	azul claro
LIGHTGREEN	10	verde claro
LIGHTCYAN	11	cian claro
LIGHTRED	12	rojo claro
LIGHTMAGENTA	13	magenta claro
YELLOW	14	amarillo
WHITE	15	blanco
BLINK	128	parpadeo

Ancho de las líneas

NORM_WIDTH	1	1 pixel de ancho
THICK_WIDTH	3	3 pixeles de ancho

Estilos de líneas

SOLID_LINE	0	Línea sólida
DOTTED_LINE	1	Línea punteada
CENTER_LINE	2	Línea centrada
DASHED_LINE	3	Línea intercalada
USERBIT_LINE	4	Estilo definido por el usuario

Valores para el último argumento de la instrucción putimage()

COPY_PUT	0	Copia directamente en la pantalla
XOR_PUT	1	Operación lógica OR exclusiva
OR_PUT	2	Operación lógica OR inclusiva
AND_PUT	3	Operación lógica AND
NOT_PUT	4	Copia lo inverso de la fuente