Capítulo

1

ELEMENTOS BÁSICOS DE PROGRAMACIÓN C

El lenguaje C++ fue creado por *Bjarne Stroustrup* [Stroustrup-2] de los laboratorios Bell y como él mismo lo afirma "C++ es un superconjunto del lenguaje de programación C". De ahí la necesidad de conocer por lo menos los aspectos más importantes de C antes de estudiar con detalle el lenguaje C++.

En este capítulo se presenta un resumen de los principales aspectos de la programación C, y está dirigido a personas que ya conocen algún otro lenguaje de programación estructurado, tal como Pascal o para aquellas personas que desean repasar el lenguaje C. Para personas que no saben del todo programar se recomienda estudiar otro texto en el que se expongan estos temas con mayor detalle, por ejemplo en [Gottfried].

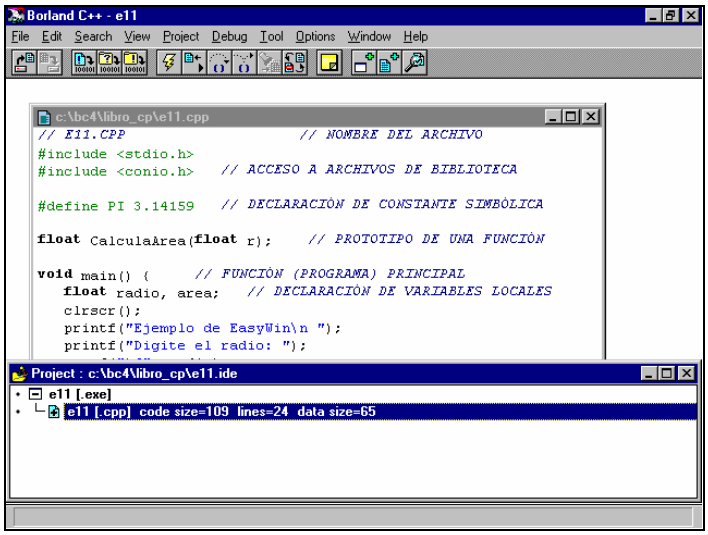
1.1 Uso del editor de Borland C++

El ambiente Borland C++ para Windows es un sistema muy sencillo de utilizar y la mejor forma de conocer todos sus secretos es consultando el manual de referencia [Borland-2] o la ayuda del sistema.

En la Figura 1.1 se presenta el editor de Borland C++ versión 4.0, y se explica en detalle la función de cada uno de los íconos del "ToolBar[1]".

Una de las características más importantes de este editor es que tiene prácticamente el manual para el programador en las ventanas de la ayuda, basta marcar una palabra clave, luego presionar el ícono de ayuda y se obtendrá una descripción completa que incluye sintaxis y ejemplos del "comando" ma

rcado.



1.2 Conceptos básicos

1 [1]Este es nombre que Borland da a la barra de íconos de acción rápida que aparece en la parte superior de la pantalla.

El lenguaje C da gran libertad al prog

ramador con respecto a la estructura del programa; sin embargo, es recomendable seguir un esqueleto de programa como el siguiente:

1. ACCESO A BIBLIOTECAS
2. DECLARACIÓN DE CONSTANTES SIMBÓLICAS
3. DECLARACIÓN DE VARIABLES GLOBALES
4. DECLARACIÓN DE PROTOTIPOS DE FUNCIONES
5. PROGRAMA PRINCIPAL (main)
6. CÓDIGO DE LAS FUNCIONES

En programación Windows, el programa principal tiene un esqueleto bastante diferente y lo estudiaremos con detalle en capítulos posteriores. Por ahora nos interesa repasar el lenguaje C estándar. Ahora bien, cuando Borland C++ para Windows se encuentra un programa C estándar, lo ejecuta utilizando el programa **EasyWin** (que viene con Borland C++). Este programa permite emular a DOS desde Windows en una ventana, de modo tal que se podrán utilizar las funciones estándar de entrada y salida de datos de C, así como funciones de pantalla de Borland C++, incluidas en versiones anteriores para DOS. Entre otras funciones se pueden citar:

```
printf(....);
scanf(....);
clrscr();
gotoxy(....);
wherex(....);
wherey(....);
```

No se podrá utilizar ninguna de las funciones gráficas de versiones anteriores de Borland C++, tales como:

```
textcolor(....);
textbackgroundcolor(....);
```

Debe quedar claro que por ahora no se están construyendo programas Windows, sino más bien estamos emulando a DOS desde Win-

dows. Claramente este no es el objetivo de este libro, pero lo hacemos únicamente con el fin de repasar el lenguaje C.

En el siguiente ejemplo se presenta un primer programa C que calcula el área de un círculo, dado su radio. En este programa se indican cada una de las partes de un programa C.

Ejemplo 1.1. Programa C para calcular el área del círculo:

```
// E11.CPP                 // NOMBRE DEL ARCHIVO
#include <stdio.h>
#include <conio.h>         // ACCESO A ARCHIVOS DE BIBLIOTECA

#define PI 3.14159         // DECLARACIÓN DE CONSTANTE SIMBÓLICA

float CalculaArea(float r);          // PROTOTIPO DE UNA FUNCIÓN

void main() {              // FUNCIÓN (PROGRAMA) PRINCIPAL
  float radio, area;       // DECLARACIÓN DE VARIABLES LOCALES
  clrscr();
  printf("Ejemplo de EasyWin\n ");
  printf("Digite el radio: ");
  scanf("%f",&radio);
  area=CalculaArea(radio);
  printf("El area es: %f",area);
}

float CalculaArea(float r) {               // CÓDIGO DE LA FUNCIÓN
 float a;                          // VARIABLE LOCAL

 a=PI*r*r;
 return a;
}
```

Para compilar y ejecutar este programa con Borland C++ 4.0 o 4.5 mediante el IDE (Integrated Development Environment) se debe crear un proyecto, para esto siga los siguientes pasos:

1. Cargue Borland C++ 4.0 o 4.5 presionando doble-click con el mouse sobre el ícono denominado Borland C++.

2. Escoja la opción **P**roject | **N**ew Project, luego la pantalla se verá como se muestra en la Figura 1.2.
3. Escoja EasyWin como su "Target Type", como se muestra en la Figura 1.2.
4. Dé click en el botón "**A**dvanced. . ." .
5. Seleccione la extensión .CPP para el nodo inicial ("initial node").
6. Dé click en el botón OK. Luego Borland C++ presentará la pantalla de la Figura 1.3.
7. Dé doble click en la entrada [.CPP] en la ventana denominada "Project".
8. Luego Borland C++ abrirá la ventana donde podrá digitar el archivo si éste no existe; si el archivo ya existe entonces Borland C++ simplemente cargará el archivo en esta ventana.
9. Para "correr" el programa seleccione la opción **D**ebug | **R**un. También puede compilar el archivo antes de ejecutarlo, seleccionando la opción **P**roject | **C**ompile.

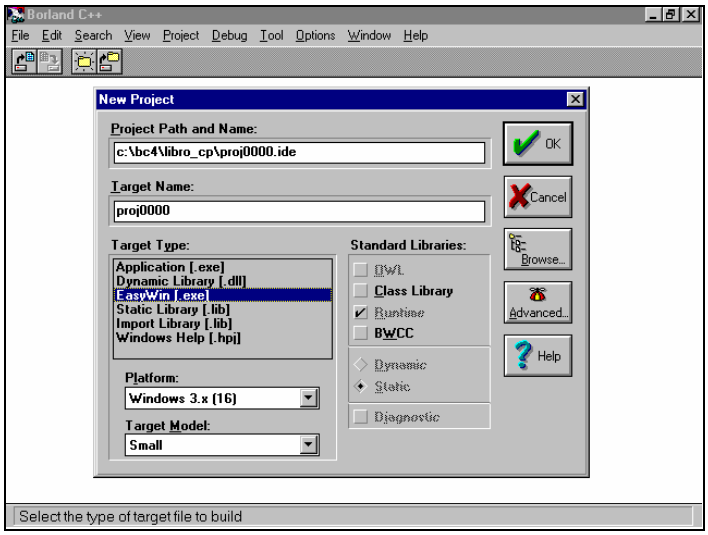
FIGURA 1.3. Ventana para administrar el

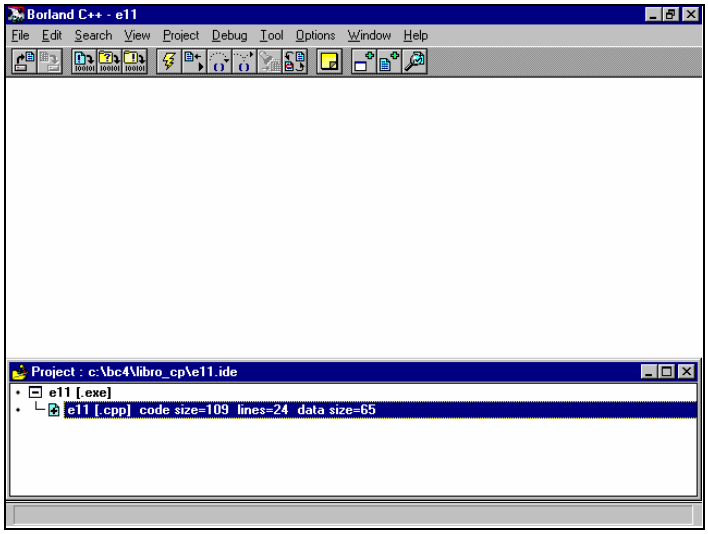
Todos los pasos anteriores se pueden efectuar de manera mucho más rápida seleccionando los íconos apropiados del "Tool Bar". Intente usted mismo usando los íconos, es fácil saber para qué sirve cada ícono ya que cuando la fecha del mouse está sobre un ícono, Borland C++ indica en la barra de ayuda (parte inferior de la pantalla) para qué sirve.

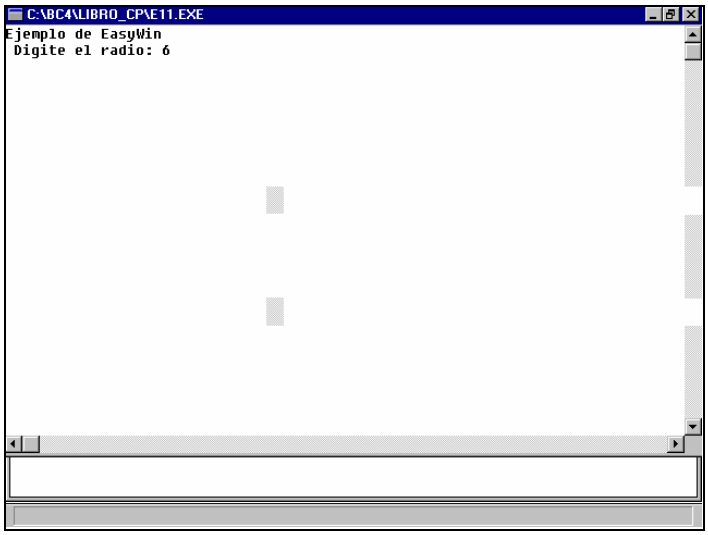
FIGURA 1.4. Ejemplo de ejecución EasyWin.

En la Figura 1.4 se muestra cómo este programa se ejecuta en una ventana que EasyWin crea para emular a DOS.

FIGURA 1.2. Creando un Proyecto nuevo.


Borland C++
File Edit Search View Project Debug Tool Options Window Help
New Project
Project Path and Name:
c:\bc4\libro_cp\proj0000.ide
Target Name:
proj0000
Target Type:
Application [.exe]
Dynamic Library [.dll]
EasyWin [.exe]
Static Library [.lib]
Import Library [.lib]
Windows Help [.hpj]
Platform:
Windows 3.x (16)
Target Model:
Small
Standard Libraries:
OWL
Class Library
Runtime
BWCC
Dynamic
Static
Diagnostic
OK
Cancel
Browse...
Advanced...
Help
Select the type of target file to build


Borland C++ - e11
File Edit Search View Project Debug Tool Options Window Help
Project : c:\bc4\libro_cp\e11.ide
e11 [.exe]
e11 [.cpp] code size=109 lines=24 data size=65



Al igual que muchos lenguajes, C posee una serie de *tipos*[2] *de datos*; a continuación se presentan los más utilizados:

Tipo	Anchura en Bits	Intervalo de valores
char	8	-128 hasta 127
int	16	-32768 hasta 32767
float	32	3.4E-38 hasta 3.4E+38
double	64	1.7E-308 hasta 1.7E+308
void	0	sin valor

El lenguanje C tiene una serie de *modificadores* de tipo que permiten cambiar el rango de valores del tipo, estos son:

signed
unsigned

2 [2]Aunque en español el separador decimal es la coma, emplearemos el punto (como en inglés) dado que el compilador así lo usa.

```
long
short
```

C posee una serie de secuencias de escape. A continuación se indican las que se usan comúnmente:

Secuencia	Propósito
\n	Nueva línea
\"	Comillas (")
\'	Apóstrofe (')
\0	Nulo

C también posee una serie de *tipos estructurados* de datos, por ahora presentaremos los arreglos. Para C un arreglo es un identificador que referencia una colección de datos, es decir un arreglo no es otra cosa que un puntero.

Por ejemplo, si en un programa C aparece una declaración y una asignación:

```
#include <string.h>        // Para usar la función strcpy
void main() {
        char N[12];
        strcpy(N, "Costa Rica");
}
```

entonces C almacena la hilera "Costa Rica" en memoria de la siguiente forma:

N[0]	N[1]	N[2]	N[3]	N[4]	N[5]	N[6]	N[7]	N[8]	N[9]	N[10]	N[11]
C	o	s	t	a		R	i	c	a	\0	

Debido a que los arreglos de C son en realidad un puntero (una dirección de memoria) se debe tener mucho cuidado con su uso, pues fá-

cilmente se pueden producir desbordamientos de memoria, es decir, que una porción de una hilera se almacene en un lugar inadecuado.

Por ejemplo si un programa C tiene la siguiente declaración y asignación:

```
#include <string.h>        // Para usar la función strcpy
void main() {
        char N[5];
        strcpy(N, "Costa Rica");
}
```

entonces C almacena la hilera "Costa Rica" en memoria de la siguiente forma:

por lo que todos los caracteres en la parte sombreada se almacenan en una posición de memoria que no se había reservado y en la que eventualmente estaban almacenados otros datos o instrucciones importantes para el programa. Este tipo de error es sumamente peligroso, pues el compilador no lo detecta y produce efectos inesperados en cualquier parte del programa.

La sintaxis de un arreglo es la siguiente:

```
tipo nombre_variable[tamaño];
```

En C, a diferencia de otros lenguajes como Pascal, todos los arreglos inician con índice cero (0).

Para la declaración de variables se debe escribir primero el tipo y luego la lista de variables. Además en C se pueden declarar variables en cualquier parte del programa. La sintaxis es la siguiente:

```
tipo lista_de_variables;
```

Por ejemplo, en un programa C se puede tener la siguiente declaración de variables:

```
int     a, b, c;
double  r, raíz;
char    ch, nombre[25];
```

Se debe tener cuidado con el nombre que se le dé a las variables, pues C distingue entre mayúsculas y minúsculas, por ejemplo, para C la variable ch es otra diferente a la variable Ch, esto porque en el primer caso la letra "C" está en minúscula y en el segundo está en mayúscula. Las variables de C se pueden clasificar en tres tipos:

➢ **Global:** cuando la variable es declarada fuera de cualquier función.
➢ **Local:** cuando la variable es declarada dentro de una función.
➢ **Parámetro Formal:** son variables que se utilizan para el paso de parámetros en una función.

Para ilustrar los diferentes tipos de variables se presenta el siguiente ejemplo. El uso de variables globales que se hace en este ejemplo no es un buen estilo de programación, sin embargo, permite ilustrar bastante bien la diferencia entre variables Locales y Globales. Para ejecutar este ejemplo cargue el proyecto E12.IDE.

🖫 **Ejemplo 1.2. El siguiente programa captura dos números enteros "a" y "b" y calcula la sumatoria de los enteros entre "a" y "b":**

```
// E12.CPP                    // NOMBRE DEL ARCHIVO
#include <stdio.h>
#include <conio.h>          // ACCESO A ARCHIVOS DE BIBLIOTECA

int suma;                   // VARIABLE GLOBAL

void CalculaSuma(int a, int b); // a y b SON PARÁMETROS FORMALES

void main() {               // FUNCIÓN (PROGRAMA) PRINCIPAL
  int valor1, valor2 ;      // DECLARACIÓN DE VARIABLES LOCALES
```

```
  clrscr();
  suma=0;              // INICIALIZA LA VARIABLE GLOBAL
  printf("Digite el valor de a: ");
  scanf("%d",&valor1);
  printf("Digite el valor de b: ");
  scanf("%d",&valor2);
  CalculaSuma(valor1, valor2);
  printf("La suma entre %d y %d es %d",valor1,valor2,suma);
}

void CalculaSuma(int a, int b) {  // CÓDIGO DE LA FUNCIÓN,
 int i;                           // CON PARÁMETROS FORMALES
 for(i=a;i<=b;i++)                // i  ES UNA VARIABLE LOCAL
    suma=suma+i;
}
```

En cuanto a la inicialización de las variables el lenguaje C es sumamente flexible, pues permite inicializar una variable en cualquier parte de un programa, incluso en el mismo lugar donde se declara la variable. La sintaxis es la siguiente:

```
tipo nombre_de_la_variable=constante;
```

por ejemplo:

```
int suma=0;
```

En código *estrictamente C++*, una variable puede ser inicializada e incluso declarada dentro de una sentencia for como se muestra en el siguiente ejemplo en el que la variable i se declara e inicializa a la vez

```
for(int i=1; i<=100; i++) {....}
```

Las constantes simbólicas son también muy utilizadas en C debido a la eficiencia con que las trata; la sintaxis es la siguiente:

```
#define nombre texto
```

donde nombre es el alias que se le da al texto. C maneja las constantes simbólicas de manera eficiente, pues en tiempo de compilación hace una sustitución literal de todas las apariciones de nombre por texto, o sea que nombre no ocupa ninguna posición de memoria. Dos ejemplos de constantes simbólicas son los siguientes:

```
#define PI  3.1415927
#define MAX  100
```

C posee una serie de operadores aritméticos, los más importantes son:

Operador	Acción
-	Resta
+	Suma
*	Multiplicación
/	División
%	Módulo de la división
--	Decremento
++	Incremento

Los operadores de incremento y decremento son únicos en C y funcionan como sigue:

x++; es equivalente a x=x+1;

x--; es equivalente a x=x-1;

Sin embargo también puede usarse ++x; o --x; pero cuál es la diferencia? Veamos el siguiente ejemplo:

```
x=10;
```

```
y=++x;
```

En este caso la variable "y" almacenará el valor entero 11, pues primero se incrementa y luego se asigna, pero si se escribe:

```
x=10;
y=x++;
```

entonces la variable "y" almacenará el valor entero 10, pues primero asigna y luego los incrementa.

C posee operadores relacionales similares a los que emplean la mayoría de los lenguajes de programación; son siguientes:

Operador	Acción
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	es igual que
!=	no es igual que

Por ejemplo, considere las siguientes expresiones:

```
x=1;
y=2;
z=(x>y);
```

entonces la variable z almacenará el valor entero 0 (cero), esto por cuanto C no posee un tipo booleano (tal como en Pascal), por lo que para C false es simplemente 0 (cero) y true es cualquier valor diferente de cero.

C también posee los operadores lógicos usuales:

Operador	Acción
&&	AND
||	OR
!	NOT

Estos operadores responden a las tablas de verdad ya conocidas para tales operadores lógicos, y típicamente aparecen en instrucciones de comparación en sentencias if, por ejemplo:

```
if((x>y) && (x<=MAX)) {....}
```

C posee una gran cantidad de operadores de asignación, entre los que se pueden mencionar los siguientes:

Operador	Acción
=	Asignación
+=	Suma el valor a variable y luego se lo asigna
-=	Resta el valor a variable y luego se lo asigna
*=	Multiplica el valor a la variable y luego se lo asigna
/*	Divide el valor entre la variable y luego se lo asigna
%=	Resto del valor de la variable y luego se lo asigna

A continuación se presentan algunos ejemplos que ilustran el uso de estos operadores:

```
`       x = 2;          // Asigna 2 a x
        x += 2;         // Suma 2 a x luego se lo asigna a x mismo
        x *= y;         // Multiplica x por y luego lo asigna a x
        x += 2*(y+z)    // Efectúa la operación de la derecha, el resultado                    // se lo sum
```

En C se llama **expresión** a cualquier combinación válida de *operadores*, *constantes* o *variables*. Son expresiones las siguientes:

```
a+b
x=y;
x<=y;
x+=y+2;
c++;
```

Un operador muy usado por los programdores C es el **operador condicional**, este es único en el lenguaje C, su sintaxis es la siguiente:

```
expresión1 ? expresión2 : expresión3
```

Veamos cómo funciona: se evalúa expresión1, si es true (es decir distinto de cero) entonces se evalúa expresión2 y este será el valor de la expresión completa, en caso contrario (o sea expresión1 da false) entonces se evalúa expresión3 y este es el resultado. En el siguiente ejemplo se utiliza el operador condicional para obtener el máximo y el mínimo entre dos números de cualquier tipo:

```
min = (a<b) ? a : b;
max = (a>b) ? a : b;
```

También se pueden usar operadores condicionales en la definición de constantes simbólicas, por ejemplo:

```
#define min(a,b)  (((a)<(b)) ? (a) : (b))
#define max(a,b)  (((a)>(b)) ? (a) : (b))
```

Para finalizar esta sección presentamos las principales funciones de entrada/salida estándar de C. Para utilizar estas funciones de biblioteca se debe incluir la biblioteca correspondiente, por ejemplo se debe incluir la biblioteca stdio.h para usar la función getchar, esto mediante la sentencia:

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

Para la entrada de caracteres uno a uno se utiliza la función **getchar** que tiene la siguiente sintaxis:

```
variable_de_carácter = getchar();
```

donde variable_de_carácter es alguna variable tipo char previamente declarada. Por ejemplo un programa C puede contener las siguientes sentencias:

```
char ch;
...
ch = getchar();
```

La función **getche()**, de la biblioeteca conio.h, funciona de manera similar a **getchar()** (tiene la misma sintaxis) y además tiene la ventaja de que no requiere presionar ENTER para que la variables sea leída.

Para visualizar un carácter la biblioteca stdio.h posee la función **putchar** cuya sintaxis es como sigue:

```
putchar(variable_de_carácter);
```

donde variable_de_carácter es alguna variable tipo char previamente declarada. Por ejemplo un programa C puede contener las siguientes sentencias:

```
char ch;
...
putchar(ch);
```

Se pueden introducir datos procedentes de dispositivos de entrada y salida mediante la función **scanf** de la biblioteca stdio.h. Esta función es sumamente potente pues se puede utilizar cualquier combinación de valores numéricos, caracteres, e hileras, la función devuelve el número de

datos que se han introducido correctamente en las variables. La sintaxis es la siguiente:

```
scanf(cadena de control, arg1, arg2, ... , argn);
```

donde cadena de control es una hilera que contiene cierta información sobre el formato de los datos y arg1, arg2, ... , argn son argumentos que representan los datos, *estos argumentos deben ser punteros* que indican las direcciones de memoria en donde se encuentran los datos.

Por ejemplo considere el siguiente fragmento de un programa:

```
#include <stdio.h>

void main() {

        float  nota1, nota2;
        int  maximo;
        char  nombre[25];
        ...
        ...
        scanf("%s %f %f %d", nombre, &nota1, &nota2, &maximo);
        ...
}
```

La cadena de control en este caso es "%s %f %f %d" . El %s indica que el primer argumento (nombre) será una hilera, el %f indica que el segundo argumento (nota1) será un valor tipo flotante, el siguiente %f indica que el tercer argumento (nota2) será también un valor tipo flotante, el %d indica que el cuarto argumento (maximo) será un valor tipo entero. Cada nombre de variable debe ir precedido del carácter &, pues los argumentos deben ser puntero, en el caso de la variable nombre no se coloca el ampersand pues un arreglo es en realidad un puntero como ya se había explicado.

Los argumentos tales como %s se conocen como caracteres de conversión, se presentan los de uso más frecuente tanto en la función scanf como en la función printf como veremos adelante.

Carácter de conversión	Significado
%d	El dato es de tipo entero
%f	El dato es de tipo flotante
%c	El dato es de tipo char
%s	El dato es una hilera
%u	El dato es un entero sin signo

Para escribir datos en la salida estándar (monitor) se utiliza la función de biblioteca **printf,** esta función puede escribir cualquier combinación de valores numéricos, caracteres o hileras. Su uso es completamente análogo al scanf con la diferencia de que su propósito es desplegar datos en vez de capturarlos. La sintaxis es la siguiente:

```
printf(cadena de control,  arg1, arg2, ... , argn);
```

donde *cadena de control* es una hilera que contiene cierta información sobre el formato de los datos y arg1, arg2, ... , argn son argumentos que representan los datos. Los argumentos pueden ser constantes, variables simples, nombres de arreglos o expresiones más complicadas, también se pueden incluir llamados a funciones. En contraste con la función scanf los argumentos de la función printf no son punteros por lo que no es necesario el uso del ampersand (&).

Por ejemplo considere el siguiente fragmento de programa:

```
#include <stdio.h>
void main() {
        float  final;
        char  nombre[25];
        ...
        ...
```

```
        printf("Nombre: %s  Promedio Final: %f\n", nombre, final );
        ...
}
```

Note que la hilera de control del printf puede incluir texto adicional y los caracteres de control pueden intercalarse con este texto. La secuencia de escape \n tiene como objetivo que una vez desplegados los datos se cambie de línea.

Para finalizar esta sección mencionaremos que la función **scanf** tiene un problema cuando se usa para leer hileras, este es, que asume el espacio en blanco como el fin de la hilera (o bien la tecla ENTER), esto genera problemas cuando pretendemos almacenar en una sola variable datos como el nombre con los apellidos. Para este tipo de datos es conveniente usar la función **gets**. Las funciones **gets** y **puts** facilitan la transferencia de hileras entre los dispositivos de entrada y salida y la computadora. La sintaxis de estas funciones es la siguiente:

```
gets(nombre_de_la_hilera);
puts(nombre_de_la_hilera);
```

Las funciones gets y puts ofrecenopciones más sencillas que printf y scanf para la lectura y escritura de hileras. Un ejemplo muy sencillo es el siguiente:

```
#include <stdio.h>
void main() {
        char nombre[40];

        printf("introduzca su nombre: ");
        gets(nombre);           // lee la hilera de teclado
        ...
        puts(nombre);           // despliega la hilera por pantalla
}
```

Otro problema de la función scanf es que con frecuencia ocasiona que algunas de las instrucciones de entrada de datos sean misteriosa-

mente saltadas; por ejemplo, en el siguiente programa la sentencia gets(N) aparentemente no es ejecutada.

```
#include <stdio.h>
void main() {
        char N[20];
        float x,y;

        scanf("%f", &x);
        gets(N);
        scanf("%f", &y);
}
```

La solución a este problema tan frecuente está en usar la función flushall() de la biblioteca stdio.h, la que se encarga de descargar todos los "streams" (corriente de caracteres) abiertos, lo cual resuelve el problema. Esta función se puede usar como se muestra en el siguiente ejemplo:

```
#include <stdio.h>
void main() {
        char N[20];
        float x,y;

        scanf("%f", &x);
        flushall();
        gets(N);
        flushall();
        scanf("%f", &y);
}
```

1.3 Sentencias de control

En esta sección presentaremos las principales sentencias de control del lenguaje C, las cuales son fundamentales en cualquier lenguaje de programación.

La sentencia while

La sentencia while se utiliza para generar bucles. Su sintaxis es la siguiente:

```
while(expresion) sentencia
```

La sentencia se ejecutará repetidamente, mientras el valor de expresión no sea cero; la sentencia puede ser simple o compuesta.

Por ejemplo, el siguiente programa despliega por pantalla todos los números enteros del 1 al 100:

```
#include <stdio.h>

void main() {
        int numero=1;

        while(numero<=100) {
                printf("%d\n",numero);
                ++numero;
        }
}
```

En el siguiente ejemplo se usa la sentencia while para calcular la media aritmética de n números reales (flotantes), el código es el siguiente:

```
#include <stdio.h>

void main() {
  int n, contador=1;
  float x, media, s
uma=0;

  printf("Cuántos números desea digitar\n");
  scanf("%d",&n);
  while(contador<=n) {
    printf("El valor de x es: ");
    scanf("%f",&x);
    suma+=x;
```

2 Elementos básicos de programación C

```
    contador++;
  }
  media=suma/n;
  printf("\nLa media aritmética es: %f",media);
}
```

La sentencia do while

La sentencia do while es similar a la sentencia while, la diferencia es que en esta la condición es chequeada al final, garantizandose así que la ejecución del programa pasará al menos una vez por el ciclo (análogo al repeat de Pascal). La sintaxis es la siguiente:

```
do sentencia while(expresion)
```

La sentencia se ejecutará repetidamente, mientras el valor de expresión no sea cero. La sentencia puede ser simple o compuesta.

Por ejemplo, para un programa que despliega por pantalla todos los números enteros del 1 al 100 presentamos una versión usando do while, el código es el siguiente:

```
#include <stdio.h>

void main() {
        int numero=1;
        do {
                printf("%d\n",numero);
                ++numero;
        }  while(numero<=100);
}
```

36 Elementos básicos de programación C

Una versión más adecuada en

un programa C es la siguiente:

```
#include <stdio.h>

void main() {
        int numero=1;
        do
            printf("%d\n",numero++);
        while(numero<=100);
}
```

En el siguiente ejemplo se usa la sentencia do while para calcular la media aritmética de n números reales (flotantes), el código es el siguiente:

```
#include <stdio.h>
void main() {
        int n, contador=1;
        float x, media, suma=0;

        printf("Cuántos números desea digitar\n");
        scanf("%d",&n);
        do {
                printf("El valor de x es: ");
                scanf("%f",&x);
                suma+=x;
                contador++;
        }  while(contador<=n);
        media=suma/n;
```

2 Elementos básicos de programación C

```
        printf("\nLa media aritmética es: %f",media);
}
```

La sentencia for

La sentencia for es el tipo de bucle más utilizado por los programadores C, esto porque el for de C es mucho más potente que sentencias for de otros lenguajes, por ejemplo el de Pascal. Su sintaxis es la siguiente:

```
for(expresion1; expresion2; expresion3) sentencia
```

en donde expresion1 se utiliza para inicializar algún parámetro que controle la repetición del bucle, expresion2 representa la condición que debe ser satisfecha para que la ejecución del ciclo continúe y expresion3 se utiliza para modificar el índice, es decir, el parámetro inicializado en expresion1. Típicamente expresion1 es una asignación, expresion2 es una expresión booleana y expresion3 es una expresión monaria o una expresión de asignación.

Una sentencia for es siempre equivalente a la siguiente expresión while:

```
expresion1;
while(expresion2) {
        sentencia;
        expresion3;
}
```

Por ejemplo, para visualizar los números enteros de 1 a 100, pero de modo tal que se desplieguen únicamente los números impares, el código es el siguiente:

```
#include <stdio.h>
void main() {
        int numero;
        for(numero=1; numero<=100; numero+=2)
                printf("%d\n",numero);
}
```

En código C++, la declaración de la variable que se va a usar como contador se puede hacer incluso dentro del mismo for; usando sintaxis C++ el ejemplo anterior se puede codificar como sigue:

```
#include <stdio.h>
void main() {
  for(int numero=1; numero<=100; numero+=2)
        printf("%d\n",numero);
}
```

Este modo C++ de utilizar la sentencia for es muy usado en lo que resta del libro, pues resulta mucho más eficiente. Sin embargo, debe tenerse claro que el compilador debe estar configurado para que compile como C++, o bien la extensión del archivo debe ser CPP.

Presentamos a continuación una versión que mediante la sentencia for del programa, calcula la media aritmética de n número reales:

```
#include <stdio.h>

void main() {
        int n;
        float x, media, suma=0;

        printf("Cuántos números desea digitar\n");
        scanf("%d",&n);
        for(int contador=1; contador<=n; contador++) {
```

4 Elementos básicos de programación C

```
            printf("El valor de x es: ");
            scanf("%f",&x);
            suma+=x;
        }
        media=suma/n;
        printf("\nLa media aritmética es: %f",media);
}
```

El bucle for de C es mucho más potente que el de otros lenguajes, por ejemplo considere el siguiente programa que calcula el promedio de n números positivos. El bucle for termina cuando se han digitado los n números o cuando el usuario digita por error un número negativo. El código es el siguiente:

```
#include <stdio.h>
void main() {
  int n;
  float x, media, suma=0;
  printf("Calcula la media aritmética de n números positivos\n\n");
  printf("Cuántos números desea digitar\n");
  scanf("%d",&n);
  for(int contador=1; ((contador<=n) && (x>=0)); contador++) {
        printf("Digite el valor de x (positivo): ");
        scanf("%f",&x);
        suma+=x;
  }
  media=suma/n;
  printf("\nLa media aritmética es: %f",media);
}
```

Nótese que el bucle for tiene una doble condición de salida, lo cual es imposible en otros lenguajes como Pascal utilizando ciclos for. Sin embargo el anterior programa tiene aún un gran defecto, y es que cuando el usuario digita por error un número negativo el programa siempre reporta el promedio. Para corregir esta situación requerimos de la sentencia condicional if-else.

La sentencia if-else

La sentencia if-else se utiliza para realizar una prueba lógica y a continuación ejecutar una de dos acciones posibles, dependiendo del resultado de esta prueba lógica. La sintaxis es la siguiente:

```
if(expresion)  sentencia1  else  sentencia2
```

en donde la parte else es optativa. La expresión se debe encontrar siempre entre paréntesis, si su valor es no nulo entonces se ejecuta la sentencia1, y si la expresión es nula entonces se ejecuta la sentencia2. Por ejemplo, el siguiente fragmento de código despliega por pantalla un número si y solamente si este es par:

```
scanf("%d",numero);
if((x%2)==0)
        printf("El número es: %d",numero);
else
        printf("El número digitado no es par");
```

En el siguiente ejemplo se presenta de nuevo el programa que calcula la media aritmética de n números positivos, pero esta vez desplegará un mensaje de error si alguno de los números digitados es negativo en lugar de desplegar la media erróneamente:

```
#include <stdio.h>
void main() {
  int n;
  float x, media, suma=0;
```

6 Elementos básicos de programación C

```
    printf("Calcula la media aritmética de n números positivos\n\n");
    printf("Cuántos números desea digitar\n");
    scanf("%d",&n);
    for(int contador=1; ((contador<=n) && (x>=0)); contador++) {
       printf("Digite el valor de x (positivo): ");
       scanf("%f",&x);
       suma+=x;
    }
    if(x>=0) {
      media=suma/n;
      printf("\nLa media aritmética es: %f",media);
    }
    else
      printf("\nERROR, se digitó un número negativo");
}
```

La sentencia switch

La sentencia switch selecciona un grupo de sentencias entre varios grupos disponibles; la sintaxis es la siguiente:

```
switch (expresion) sentencia
```

donde expresión devuelve un valor entero o bien es de tipo char. La sentencia incluida es generalmente una sentencia compuesta que especifica posibles opciones por seguir, cada opción especifica una o más sentencias individuales. Ahora bien, cada opción va precedida de la palabra reservada case, con la siguiente sintaxis general:

```
case expresion:
        sentencia1;
```

```
        sentencia2;
    ....
    sentencian;
```

donde expresión representa una expresión constante de valor entero. Considere el siguiente segmento de programa:

```
switch (color = getchar()) {
case 'r':
case 'R':
        printf("COLOR ROJO");
        break;
case 'b':
case 'B':
        printf("COLOR BLANCO");
        break;
case 'a':
case 'A':
        printf("COLOR AZUL");
        break;
}
```

por lo tanto presentará COLOR ROJO si la escogencia es 'r' o 'R', presentará COLOR BLANCO si la escogencia es 'b' o 'B' y presentará COLOR AZUL si la escogencia es 'a' o 'A'. Note el uso de la sentencia break, cuya función es transferir el control de la ejecución del programa fuera de la sentencia switch.

Una versión mejorada del anterior código es la siguiente:

```
switch (color = toupper(getchar())) {
case 'R':
        printf("COLOR ROJO");
        break;
case 'B':
        printf("COLOR BLANCO");
        break;
case 'A':
        printf("COLOR AZUL");
        break;
```

8 Elementos básicos de programación C

```
        default:
                printf("ERROR");
        }
```

Decimos que es una versión mejorada por dos razones, primero porque la función de biblioteca toupper convierte automáticamente en mayúscula un carácter digitado por el usuario, por lo que se hace innecesarias las expresiones case 'r', case 'b' y case 'a'. Segundo, por el uso de la etiqueta default que permite agrupar un conjunto de sentencias que se ejecutarán si ninguna de las etiquetas coincide con expresión.

La sentencia switch es muy utilizada para implementar menús en un programa, por ejemplo considere el programa que incluye una función (en la siguiente sección se presenta con detalle el uso de funciones en C) que despliega un menú y retorna la opción escogida, para que luego el programa, mediante una sentencia switch, ejecute las acciones seleccionas por el usuario:

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

char menu();                    // PROTOTIPO
void main() {
   char escogencia=menu();
   clrscr();
   switch(escogencia) {
        case '1': printf("Se ejecutaría la función Leer_datos"); break;
        case '2': printf("Se ejecutaría la función Borrar_datos"); break;
        case '3': printf("Se ejecutaría la función Modificar_datos");break;
        case '4': printf("Se ejecutaría la función Buscar_datos"); break;
```

```
            case '5': printf("Se ejecutaría la función Calcular_Sueldo"); break;
        case '6': printf("Se ejecutaría la función Imprimir_cheque"); break;
        case '7': break;
        default: printf("ERROR al escoger la opción");
    }
}

char menu() {
    char opcion;
    printf("[1] Leer datos del empleado\n");
    printf("[2] Borrar datos del empleado\n");
    printf("[3] Modificar datos del empleado\n");
    printf("[4] Buscar datos del empleado\n");
    printf("[5] Calcular sueldo del empleado\n");
    printf("[6] Imprimir cheque del empleado\n");
    printf("[7] Menú anterior\n");
    opcion= toupper(getche());
    return opcion;
}
```

1.4 Funciones y recursividad

En esta sección se discuten los aspectos básicos sobre funciones y funciones recursivas. El lenguaje C a diferencia de otros lenguajes no tiene procedimientos y funciones, sino solamente funciones. Un procedimiento se implementa mediante una función que devuelve un "dato" de tipo void.

La sintaxis general de una función es la siguiente:

```
tipo nombre_de_la_función(declaración de parámetros)  {
        cuerpo de la función
}
```

Si no se especifica ningún tipo entonces se supone que la función, por defecto, proporcionará un valor entero.

🖫 Ejemplo 1.3

10 Elementos básicos de programación C

Para ejecutar este ejemplo cargue el proyecto E13.IDE. En este ejemplo se presenta un programa que contiene la función Potencia la cual permite calcular x^n para $x \geq 0$ real y *n* entero, la función devuelve -1 si *x* es negativo:

```
//  E13.CPP
#include <stdio.h>
#include <conio.h>          // ACCESO A ARCHIVOS DE BIBLIOTECA

float Potencia(float x,int n); // PROTOTIPO DE UNA FUNCIÓN

void main() {                  // FUNCIÓN (PROGRAMA) PRINCIPAL
  float base;
  int exponente;              // DECLARACIÓN DE VARIABLES LOCALES

  clrscr();
  printf("Digite la base: ");
  scanf("%f",&base);
  printf("Digite el exponente: ");
  scanf("%d",&exponente);
  printf("El valor de %f elevado a la %d es: %f",
         base,exponente,Potencia(base,exponente));
}

float Potencia(float x,int n) {  // CÓDIGO DE LA FUNCIÓN
  float exp=1;
  if(x<0)
    return -1;
  else
    for(int i=1; i<=n; i++)
          exp*=x;
```

```
    return exp;
}
```

La sentencia return tiene dos aplicaciones importantes. En primer lugar produce una salida inmediata del curso de la función, esto es el return hace que la ejecución del programa vuelva al código que hizo la llamada (a la línea siguiente). En segundo lugar se puede utilizar para proporcionar un valor, tal como se hizo en el ejemplo anterior.

Para implementar un procedimiento se utiliza una función que no proporciona ningún valor, es decir se declara de tipo void. Por ejemplo el siguiente procedimiento imprime los números enteros pares de 1 al n donde n se recibe como parámetro:

```
void imprime(int n) {
        for(int i=2;i<=n;i+=2)
                printf("%d\n",i);
}
```

Sin embargo, los programadores en C utilizan con más frecuencia las funciones que los procedimientos, pues aunque se tenga un procedimiento generalmente se acostumbra que retorne 1 para indicar que el procedimiento se ejecutó con éxito y 0 (cero) en caso contrario.

Como se mostró en el ejemplo 1.3 la función Potencia requirió de un **prototipo** que se ubica antes de la función main(). Los prototipos

tienen dos objetivos fundamentales. Primero identificar el tipo que retorna la función para que el compilador pueda generar el código correcto para el tipo de datos que proporciona la función. En segundo lugar especifica el número y el tipo de argumentos que emplea la función impidiendo que la función se invoque con un tipo o número equivocado de argumentos. La sintaxis de un prototipo es la siguiente:

```
tipo nombre_de_la_función(lista_de_parámetros);
```

por ejemplo:

```
double suma(double a, double b);
```

Usualmente los prototipos se colocan al principio del programa, o en un archivo de encabezados (prototipos) con extensión .h o .hpp que se incluye en el programa mediante un #include "nombre_del_archivo".

Los prototipos de C no son parte del C original de Kernighan y Ritchie, sino que han sido agregados por el comité de estándares ANSI, cuyo objetivo es dotar a C de una comprobación estricta de tipos, similar a la que ofrecen algunos lenguajes como Pascal.

Llamadas por valor y por referencia

En general, se pueden pasar argumentos a las subrutinas de dos formas: la primera se conoce como paso de parámetros *por valor*. Este método consiste en copiar el valor de un argumento en un parámetro formal de la subrutina, por lo tanto los cambios que se hagan en los parámetros de la subrutina no tienen efecto en las variables utilizadas para hacer la llamada.

La segunda forma se conoce como paso de parámetros *por referencia,* en este método la *dirección* del argumento se copia en el parámetro, de modo tal que dentro de la subrutina se utiliza esta dirección para tener acceso al argumento real que se ha utilizado en la llamada. Por lo tanto los cambios que se hagan en el parámetro afectarán o modificarán la variable utilizada para llamar la subrutina.

C utiliza únicamente paso de parámetros por valor, sin embargo se puede "forzar" para que las llamadas sean por referencia. Esto se logra mediante la utilización de punteros a las variables (en la siguiente sección se expone el tema de punteros con mayor detalle). Por ejemplo considere el siguiente programa que captura dos números enteros e intercambia sus valores:

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

```
    void intercambiar(int *x,int *y); // PROTOTIPO

void main() {
        int a, b;

        printf("Digite los valores de A y B\n");
        scanf("%d %d", &a, &b);
        intercambiar(&a,&b);
        printf("Los nuevos valores de A y B son %d %d",a,b);
}
void intercambiar(int *x,int *y)  {
        int tempo;

        tempo=*x;       // Guarda el valor que está en la dirección de x
        *x=*y;          // Guarda y en x
        *y=tempo;       // Guarda x en y
}
```

El operador * se utiliza para accesar la variable a la cual apunta su operando, por lo tanto en el procedimiento intercambiar los parámetros son en realidad un puntero a x y otro a y. Por lo tanto cuando este procedimiento se invoca se debe hacer con las *direcciones de los argumentos*, por esta razón el llamado al procedimiento intercambiar se hace con el operador &.

Recursividad

Matemáticamente se dice que una función es recursiva si ésta se define en términos de sí misma. Computacionalmente se dice que una función es recursiva si en una sentencia situada en el cuerpo de la función se invoca a sí misma. El lenguaje C permite la implementación de funciones recursivas.

Por ejemplo, supóngase que se desea implementar la función potencia $f(x) = x^n$ en forma recursiva. Lo primero que se debe hacer es definir una relación de recurrencia equivalente a *f*. Esta relación de recurrencia es la siguiente:

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x * x^{n-1} & \text{si } n \geq 1 \end{cases}$$

escrita de una manera más cercana a los lenguajes de programación, se tiene:

$$\text{Potencia}(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ x * \text{Potencia}(x, n-1) & \text{si } n >= 1 \end{cases}$$

El siguiente programa contiene el código C de la función potencia en forma recursiva:

```
#include <stdio.h>
#include <conio.h>          // ACCESO A ARCHIVOS DE BIBLIOTECA

float Potencia(float x,int n); // PROTOTIPO DE UNA FUNCIÓN

void main() {                  // FUNCIÓN (PROGRAMA) PRINCIPAL
  float base;
  int exponente;              // DECLARACIÓN DE VARIABLES LOCALES

  clrscr();
  printf("Digite la base: ");
  scanf("%f",&base);
  printf("Digite el exponente: ");
  scanf("%d",&exponente);
  printf("El valor de %f elevado a la %d es: %f"
                  base,exponente,Potencia(base,exponente));
}

float Potencia(float x,int n) {  // CÓDIGO DE LA FUNCIÓN
  if(n==0)
    return 1;
  else
    return x*Potencia(x,n-1);
}
```

C también permite implementar funciones con recursión doble o recursión de árbol, por ejemplo considere la sucesión de Fibonacci definida como sigue:

$$F_n = \begin{cases} 1 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ F_{n-1} + F_{n-2} & \text{si } n > 2 \end{cases}$$

se puede implementar en C como sigue:

```
int Fibonacci(int n) {
  if((n==1) || (n==2))
    return 1;
  else
    return Fibonacci(n-1)+Fibonacci(n-2);
}
```

1.5 Arreglos y compilación separada

La sintaxis de los arreglos (vectores) de C es la siguiente:

```
tipo nombre_variable[tamaño];
```

donde el tipo especifica el tipo de datos de cada uno de los elementos que constituyen el arreglo. Mientras que tamaño define cuántos elementos puede contener el arreglo o vector. Por ejemplo la siguiente sentencia declara un arreglo de 50 elementos de tipo flotante:

```
float notas[50];
```

En C todos los arreglos tienen como primer índice el cero, así el arreglo anterior tendrá 50 elementos, desde notas[0] hasta notas[49].

Como se había mencionado en la sección 1.1 se debe tener sumo cuidado con el manejo de arreglos en C, pues C **no** controla el tamaño de los arreglos, es completa responsabilidad del programador garantizar que no asignará valores en campos de un arreglo sobrepasando su tama-

ño. Por ejemplo, las siguientes instrucciones son completamente incorrectas:

```
float notas[50];
for(int i=0; i<=100; i++)   // INCORRECTO
        notas[i]=1;
```

Lo anterior es incorrecto por cuanto asigna el valor 1 a campos del arreglo para los cuales no se ha reservado memoria, es decir, del campo

notas[50] al campo notas[100]. Un error de este tipo provoca errores inesperados en el programa, dado que se han asignado valores en posiciones de memoria donde podría estar parte del código del programa.

Las cadenas o hileras ("strings") en C se implementan mediante arreglos de caracteres. Por ejemplo:

```
char nombre[40];
```

declara una hilera de 39 campos, pues el último campo será ocupado por el carácter nulo "\0".

Para el manejo de hileras, C posee una serie de funciones de biblioteca muy utilizadas, por ejemplo: strcpy(), strcat(), strlen(), strcmp(). Para mayor detalle sob

re estas funciones puede consultarse el manual.

En C se pueden pasar arreglos como parámetros en una función de manera muy simple, no es necesario definir un tipo de datos nuevo como en Pascal. Para pasar un arreglo como parámetro a una función, el nombre del arreglo debe aparecer solo, sin corchetes o índices, como un argumento actual de la función. Por ejemplo el siguiente programa captura desde el teclado un arreglo de enteros, para luego encontrar el máximo de los elementos en el arreglo. Para ejecutar este ejemplo cargue el proyecto E14.IDE.

🖫 Ejemplo 1.4

```
     // E14.CPP
#include <stdio.h>
#include <conio.h>          // ACCESO A ARCHIVOS DE BIBLIOTECA

#define MAX 100

void captura(int *n, int x[MAX]);  // PROTOTIPOS DE FUNCIONES
int  maximo(int n, int x[MAX]);

void main() {                  // FUNCIÓN (PROGRAMA) PRINCIPAL
  int tamano, vector[MAX];
  captura(&tamano,vector);
  printf("El valor máximo es %d",maximo(tamano,vector));
}

void captura(int *n, int x[MAX]) {
  int tempo;
  printf("Digite el tamaño del vector (menor a 100)\n");
  scanf("%d",&tempo);
  *n=tempo;
  for(int i=0; i<tempo;i++) {
     printf("Digite la entrada %d\n",i);
     scanf("%d",&x[i]);
  }
}

int  maximo(int n, int x[MAX]) {
 int max=x[0];
 for(int i=1; i<n;i++) {
    if(x[i]>max)
       max=x[i];
 }
 return max;
}
```

Nótese que en la función void captura(int *n, int x[MAX]) el parámetro n está precedido por un * para indicar que es en realidad un puntero a un entero, lo cual permite que el parámetro sea por referencia. Esto se implementa así porque el valor de n debe ser conocido por la función

máximo. La función captura puede implementarse como sigue, para evitar el uso de la variable temporal tempo.

```
void captura(int *n, int x[MAX]) {
  printf("Digite el tamaño del vector (menor a 100)\n");
  scanf("%d",n);
  for(int i=0; i<*n;i++) {
    printf("Digite la entrada %d\n",i);
    scanf("%d",&x[i]);
  }
}
```

C también permite los arreglos multidimensionales (matrices). La sintaxis es la siguiente:

```
tipo nombre_de_la_matriz[tamaño1][tamaño2]...[tamañoN];
```

donde, al igual que en los arreglos, todos los índices comienzan en cero (0). Los arreglos más comúnmente utilizados son las matrices bidimensionales. En el siguiente ejemplo se presenta un programa que captura dos matrices de números enteros y calcula su suma, para luego desplegar por pantalla el resultado. Si desea ejecutar este ejemplo, cargue el proyecto E15.IDE.

🖫 Ejemplo 1.5

```
// E15.CPP
#include <stdio.h>
#include <conio.h>          // ACCESO A ARCHIVOS DE BIBLIOTECA

#define MAX 25              // TAMAÑO MÁXIMO DE LAS MATRICES

void captura(int *n,int *m,int x[MAX][MAX]);
void imprime(int n, int m,int x[MAX][MAX]);
void suma(int n, int m,int x[MAX][MAX],int y[MAX][MAX],int s[MAX][MAX]);

void main() {                   // FUNCIÓN (PROGRAMA) PRINCIPAL
  int fil1, col1, fil2, col2, M1[MAX][MAX], M2[MAX][MAX], S[MAX][MAX];

  captura(&fil1,&col1,M1);
```

```
        captura(&fil2,&col2,M2);
  if((fil1==fil2) && (col1==col2)) {
    suma(fil1,col1,M1,M2,S);
    imprime(fil1,col1,S);
  }
  else
    printf("ERROR en el tamaño de la matrices");
}

void captura(int *n,int *m,int x[MAX][MAX]) {
  clrscr();
  printf("Digite el número de filas (menor a 25)\n");
  scanf("%d",n);
  printf("Digite el número de columnas (menor a 25)\n");
  scanf("%d",m);
  for(int i=1;i<=*n;i++) {
    for(int j=1;j<=*m;j++) {
          gotoxy(4*j,i+4);
          scanf("%d",&x[i][j]);
    }
  }
}

void imprime(int n,int m,int x[MAX][MAX]) {
  clrscr();
  printf("La matriz es\n\n");
  for(int i=1;i<=n;i++) {
    for(int j=1;j<=m;j++) {
          gotoxy(4*j,i+4);
          printf("%d",x[i][j]);
    }
  }
}

void suma(int n, int m,int x[MAX][MAX],int y[MAX][MAX],int s[MAX][MAX]) {
  for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
          s[i][j]=x[i][j]+y[i][j];
}
```

En el siguiente ejemplo se pretende ilustrar cómo Borland C++ facilita la compilación separada. Para esto se construirá un programa que tenga el siguiente menú principal:

[1] Operaciones con vectores
[2] Operaciones con matrices
[3] Salir

si el usuario escoge [1] entonces despliega el siguiente menú:

[1] Máximo de un vector
[2] Mínimo de un vector
[3] Menú anterior

pero si el usuario escoge [2] entonces despliega el siguiente menú:

[1] Múltiplo escalar de una matriz
[2] Suma de matrices
[3] Menú anterior

La idea es que exista un archivo denominado VECTOR.HPP donde estén los prototipos de las funciones para vectores y un archivo VECTOR.CPP donde esté el código, un archivo MATRIZ.HPP y MATRIZ.CPP que contendrán los prototipos y los códigos respectivamente de las operaciones con matrices, también debe existir un archivo MENU.HPP y MENU.CPP. Finalmente un archivo E16.CPP donde está el código del programa principal, es decir, la función main(). Estos archivos se presentan a continuación:

🖫 Ejemplo 1.6

```
// E16.CPP
#include "menu.hpp"
#include "matriz.hpp"
#include "vector.hpp"
#include <conio.h>
#include <stdio.h>

void main() {
```

```
      char opcion1;
do {
  clrscr();
  opcion1=menu();
  clrscr();
  switch(opcion1) {
       case '1': clrscr();
                 char opcion2=menu_vector();
                 switch(opcion2) {
                   case '1': int tamano, vector[MAX];
                   captura(&tamano,vector);
                             printf("El valor máximo es                    %d",maxim
                             getche();
                             clrscr();
                             break;
                   case '2': captura(&tamano,vector);
                             printf("El valor mínimo es                    %d",minim
                             getche();
                             clrscr();
                   break;
                 }
                 break;
       case '2': clrscr();
                 char opcion3=menu_matriz();
                 switch(opcion3) {
                   case '1':
                          int fil1, col1, fil2, col2, M1[MAX][MAX],          M2[MAX][M
                            captura(&fil1,&col1,M1);
                          captura(&fil2,&col2,M2);
                            if((fil1==fil2) && (col1==col2)) {
                               suma(fil1,col1,M1,M2,S);
                               imprime(fil1,col1,S);
                             }
                          else
                               printf("ERROR en el tamaño de la matrices");
                          getche();
                          clrscr();
                          break;
                   case '2': captura(&fil1,&col1,M1);
                             multiplo_escalar(fil1,col1,6,M1,S);
                             imprime(fil1,col1,S);
```

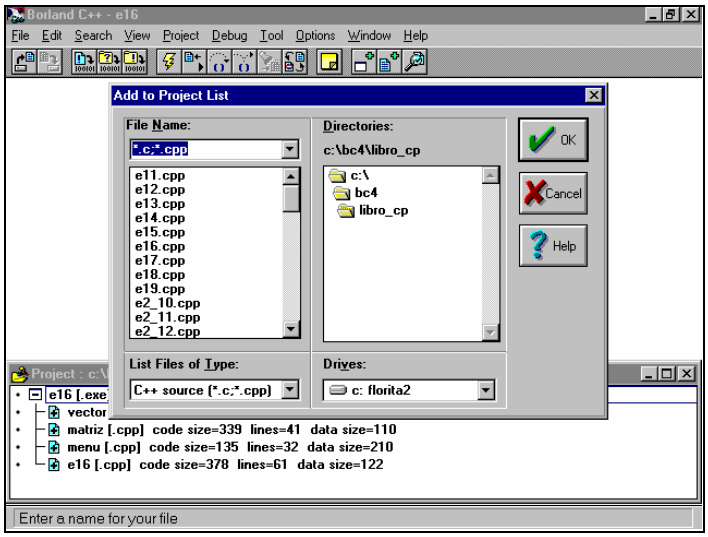
```
                                    getche();
                                clrscr();
                    break;
                   }
                   break;
          case '3': break;
          default: printf("ERROR al escoger la opción");
     }
   } while(opcion1!='3');
}

// vector.hpp
#define MAXVEC 100

// PROTOTIPOS DE FUNCIONES
void captura(int *n, int x[MAXVEC]);
int  maximo(int n, int x[MAXVEC]);
int  minimo(int n, int x[MAXVEC]);

// vector.cpp
#include <stdio.h>
#include <conio.h>          // ACCESO A ARCHIVOS DE BIBLIOTECA
#include "vector.hpp"

void captura(int *n, int x[MAXVEC]) {
   int tempo;
   clrscr();
   printf("Digite el tamaño del vector (menor a 100)\n");
   scanf("%d",&tempo);
   *n=tempo;
   for(int i=0; i<tempo;i++) {
      printf("Digite la entrada %d\n",i);
      scanf("%d",&x[i]);
   }
}

int  maximo(int n, int x[MAXVEC]) {
  int max=x[0];
  for(int i=1; i<n;i++) {
     if(x[i]>max)
        max=x[i];
```

```
        }
  return max;
}

int  minimo(int n, int x[MAXVEC]) {
  int max=x[0];
  for(int i=1; i<n;i++) {
     if(x[i]<max)
        max=x[i];
  }
  return max;
}

// matriz.hpp
#define MAX 25

// PROTOTIPOS DE FUNCIONES
void captura(int *n,int *m,int x[MAX][MAX]);
void imprime(int n, int m,int x[MAX][MAX]);
void suma(int n, int m,int x[MAX][MAX],int y[MAX][MAX],int s[MAX][MAX]);
void multiplo_escalar(int n, int m,int k,int x[MAX][MAX],int s[MAX][MAX]);

// matriz.cpp
#include <stdio.h>
#include <conio.h>
#include "matriz.hpp"

void captura(int *n,int *m,int x[MAX][MAX]) {
   clrscr();
   printf("Digite el número de filas (menor a 25)\n");
   scanf("%d",n);
   printf("Digite el número de columnas (menor a 25)\n");
   scanf("%d",m);
   for(int i=1;i<=*n;i++) {
     for(int j=1;j<=*m;j++) {
            gotoxy(4*j,i+4);
            scanf("%d",&x[i][j]);
     }
   }
}
```

```
    void imprime(int n,int m,int x[MAX][MAX]) {
  clrscr();
  printf("La matriz es\n\n");
  for(int i=1;i<=n;i++) {
    for(int j=1;j<=m;j++) {
          gotoxy(4*j,i+4);
          printf("%d",x[i][j]);
    }
  }
}

void suma(int n, int m,int x[MAX][MAX],int y[MAX][MAX],int s[MAX][MAX]) {
  for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
          s[i][j]=x[i][j]+y[i][j];
}

void multiplo_escalar(int n, int m,int k,int x[MAX][MAX],int s[MAX][MAX]) {
  for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
          s[i][j]=k*x[i][j];
}

// menú.hpp
// PROTOTIPOS
char menu();
char menu_vector();
char menu_matriz();

// menú.cpp
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include "menu.hpp"

char menu() {
   char opcion;
   printf("[1] Operaciones con vectores\n");
   printf("[2] Operaciones con matrices\n");
   printf("[3] Salir\n");
   opcion= toupper(getche());
```

FIGURA 1.5. Generador de Proyectos de Bor-

```
        return opcion;
}

char menu_vector() {
   char opcion;
   printf("[1] Máximo de un vector\n");
   printf("[2] Mínimo de un vector\n");
   printf("[3] Menú anterior\n");
   opcion= toupper(getche());
   return opcion;
}

char menu_matriz() {
   char opcion;
   printf("[1] Suma de matrices\n");
   printf("[2] Producto por un escalar\n");
   printf("[3] Menú anterior\n");
   opcion= toupper(getche());
   return opcion;
}
```



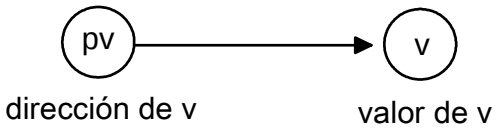
Para ejecutar este ejemplo cargue el proyecto E16.IDE, nótese que en este se incluyen todos los archivos con extensión .CPP correspondientes a cada uno de los módulos. Esto se muestra en la Figura 1.5. Si no tiene el archivo E16.IDE, entonces cree el proyecto tal como se explica en la sección 1.2 pero incluya todos los archivos .CPP.

1.6 Punteros y estructuras

Supongamos que v es una variable que representa un determinado dato. El compilador automáticamente asigna celdas de memoria para este dato. El dato puede ser accesado solamente si conocemos su localización (*dirección*), es decir la localización de la primera celda. La dirección de v puede ser determinada mediante el *operador de dirección* &, usando para esto la expresión &v. Así por ejemplo si se tiene la expresión:

```
pv = &v
```

entonces la nueva variable pv es un puntero a v, es decir pv representa la *dirección* de v y no su valor. El dato representado por v puede ser accesado por la expresión *pv, donde el operador * se llama operador de *indirección*, el cual opera solamente sobre variables de tipo puntero. Gráficamente:



de donde se deducen las siguientes relaciones:

```
pv = &v  y  v = *pv
```

Si una variable va a contener un puntero, entonces es preciso declararla como tal. La sintaxis C para declarar punteros es la siguiente:

```
tipo  *nombre_variable;
```

donde tipo puede ser cualquier tipo válido en C y nombre_variable es el nombre de la variable puntero. Por ejemplo, la siguiente sentencia declara una variable a un entero:

```
int *tempo;
```

FIGURA 1.6. Valor de u, v y de los punteros pu, pv

A continuación s

e presenta un programa muy simple que utiliza punteros:

```
#include <stdio.h>

void main() {
  int u=5;
  int v;
  int *pv;          // declara un puntero a un entero
  int *pu;          // declara un puntero a un entero

  pu = &u;          // asigna la dirección de u a pu
  v = *pu;          // asigna el valor de u a v
  pv = &v;          // asigna la dirección de v a pv

  printf("u=%d   *pu=%d   &u=%X  pu=%X",u,*pu,&u,pu);
  printf("\n\n");
  printf("v=%d   *pv=%d   &v=%X  pv=%X",v,*pv,&v,pv);
}
```

cuya salida se presenta en la Figura 1.6:

En el siguiente ejemplo ilustramos nuevamente el uso de parámetros por valor y por referencia, esto debido a que la comprensión de estos conceptos es central en programación C (y en cualquier otro lenguaje).

```
#include <stdio.h>
void prueba1(int u, int v);          // Parámetros por valor
void prueba2(int *pu, int *pv);      // Parámetros por referencia (estilo C)
```

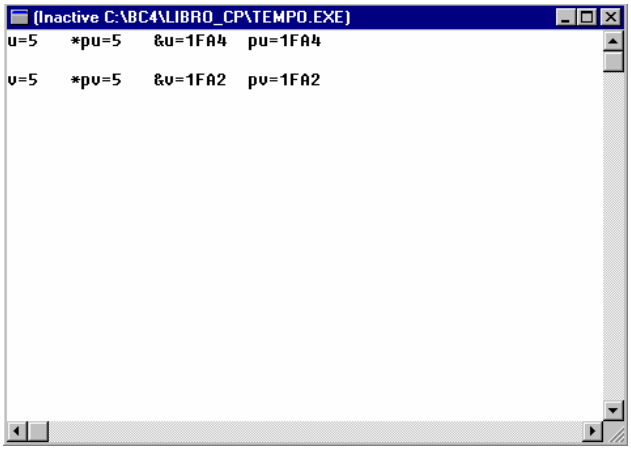
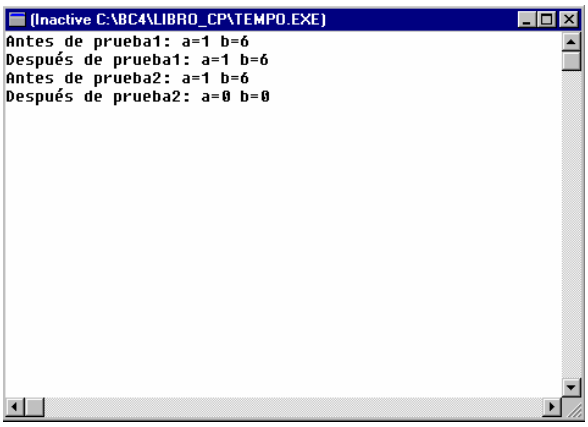


FIGURA 1.7.
Salida de las funciones prueba1 y prueba2

```
void main() {
  int a=1;
  int b=6;

  printf("Antes de prueba1: a=%d b=%d",a,b);
  printf("\n");
  prueba1(a,b);
  printf("Después de prueba1: a=%d b=%d",a,b);
  printf("\n");
  printf("Antes de prueba2: a=%d b=%d",a,b);
  printf("\n");
  prueba2(&a,&b);
  printf("Después de prueba2: a=%d b=%d",a,b);
}

void prueba1(int u, int v) {
  u=0;
  v=0;
}

void prueba2(int *pu, int *pv) {
  *pu=0;
  *pv=0;
}
```

cuya salida se presenta en la Figura 1.7.



Como puede verse, los valores de las variables a y b **no** "salen" alterados de la función prueba1 puesto que en esta función los parámetros son por valor. Mientras que los valores de las variables a y b **sí** "salen" alterados de la función prueba2, dado que en esta función los parámetros son punteros a las variables, es decir, las variables son pasadas por referencia.

C++ acepta la sintaxis del ejemplo anterior, sin embargo, esta es sumamente incómoda y de bajo nivel. Debido a esto C++ posee un nuevo mecanismo para generar parámetros por referencia en forma automática, este mecanismo consiste en preceder el parámetro de la función con un &, de modo tal que el & cumple una tarea similar a la palabra reservada VAR en el lenguaje de programación Pascal, cuando se usa para declarar parámetros por referencia. A continuación se reformula el ejemplo anterior para usar sintaxis C++. Para que este programa se ejecute adecuadamente, asegúrese de estar compilando como C++, esto se consigue colocando extensión .CPP al archivo o bien configurando el sistema para que compile siempre como C++, esto mediante el menú Options del editor de Borland C++.

```
    #include <stdio.h>
void prueba1(int u, int v);          // Parámetros por valor
void prueba2(int &u, int &v);        // Parámetros por referencia (estilo C++)

void main() {
  int a=1;
  int b=6;


  printf("Antes de prueba1: a=%d b=%d",a,b);
  printf("\n");
  prueba1(a,b);
  printf("Después de prueba1: a=%d b=%d",a,b);
  printf("\n");
  printf("Antes de prueba2: a=%d b=%d",a,b);
  printf("\n");
  prueba2(a,b);
  printf("Después de prueba2: a=%d b=%d",a,b);
}

void prueba1(int u, int v) {
  u=0;
  v=0;
}

void prueba2(int &u, int &v) {
  u=0;
  v=0;
}
```

La salida de este programa es exactamente la misma que la del ejemplo anterior presentada en la Figura 1.7.

Punteros y arreglos

Ya hemos mencionado que existe un parentesco muy cercano entre punteros y arreglos. En realidad se pueden manipular los arreglos vía aritmética de punteros. Esta aritmética permite solamente dos operaciones, a saber, suma y resta. Así por ejemplo si se tiene la siguiente declaración:

```
int x[10];
```

es decir, si x es un arreglo de enteros, entonces son equivalentes las siguientes expresiones:

x[i] es equivalente a *(x+i)
&x[i] es equivalente a x+i

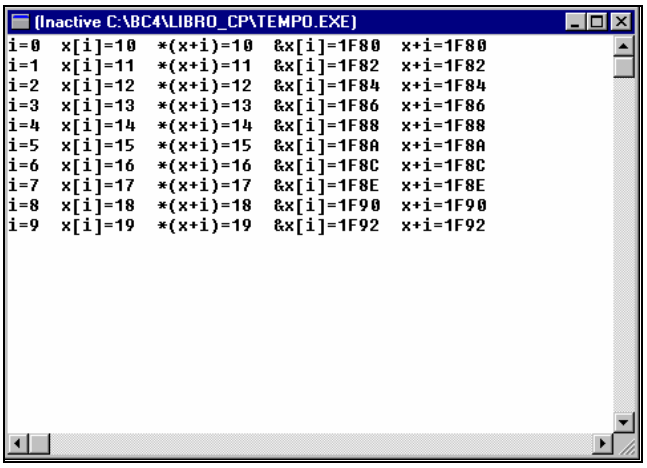
De donde se concluye que x es un puntero que está direccionado o apuntando a la primera casilla de memoria, en donde está almacenado el arreglo. Para convencerse de esto analice la salida del siguiente programa:

FIGURA 1.8. Salida mediante arreglos y punteros.

```
#include <stdio.h>

void main() {
  int x[10];
  int i;

for(i=0;i<=9;i++)
  x[i] = i+10;
for(i=0;i<=9;i++)
  printf("i=%d  x[i]=%d  *(x+i)=%d  &x[i]=%X  x+i=%X\n",i,x[i],*(x+i),&x[i],x+i);
}
```

que se presenta en la Figura 1.8.

Debido a que un arreglo bidimensional es en realidad un arreglo de arreglos es natural pensar que tales arreglos también puedan representarse en términos de punteros. En efecto este es el caso, sin embargo del hecho de que un arreglo bidimensional sea un arreglo de arreglos se puede deducir inmediatamente cómo es la correspondiente representación por medio de punteros. Dejamos al lector investigar sobre tal representación.



Estructuras

En C una *estructura* es una colección de variables a las cuales se hace alusión mediante un único nombre, permitiendo agrupar en forma cómoda un conjunto de datos relacionados. C++ generaliza el concepto de estructura al concepto de *clase*. La sintaxis de una estructura es la siguiente:

```
struct  nombre_tipo_estructura {
            tipo nombre_elemento1;
            tipo nombre_elemento2;
            tipo nombre_elemento3;
                    .........
            tipo nombre_elementoN;
};
```

Por ejemplo, la siguiente estructura agrupa los datos de un empleado:

```
struct empleado {
            char cedula[9];
            char nombre[40];
            int ciudad;
```

```
};
```

Ahora para declarar variables de este tipo, se hace como sigue:

```
struct empleado empleado1, empleado2;
```

En esta expresión se declararon dos variables empleado1 y empleado2 de tipo empleado. Ahora bien, es posible combinar en una sola expresión la composición de la estructura con la declaración de las variables. Para esto la sintaxis es la siguiente:

```
struct  nombre_tipo_estructura {
        tipo nombre_elemento1;
        tipo nombre_elemento2;
        tipo nombre_elemento3;
                .........
        tipo nombre_elementoN;
} variable1, variable2, ... , variableN;
```

En el siguiente ejemplo se resumen en una sola sentencia los últimos dos ejemplos:

```
struct empleado {
        char cedula[9];
        char nombre[40];
        int ciudad;
} empleado1, empleado2;
```

También es posible declarar arreglos de estructuras. Por ejemplo la siguiente sentencia declara un arreglo de 100 empleados:

```
struct empleado {
        char cedula[9];
        char nombre[40];
        int ciudad;
} empleados[100];
```

Los elementos de una estructura se procesan generalmente como entidades separadas, por lo que C provee un mecanismo para tener acceso a miembros individuales de una estructura. Esto se logra con el operador de selección, es decir, el punto (.); la sintaxis es la siguiente:

```
variable.miembro
```

donde variable se refiere al nombre de una variable de tipo estructura y miembro es el nombre de un campo particular de la estructura. Por ejemplo, en el siguiente fragmento de código se declaran variables de tipo empleado, que es una estructura, y se hacen asignaciones con sus miembros:

```
struct empleado {
        char cedula[9];
        char nombre[40];
        int ciudad;
} empleado1, empleado2;

int x=empleado1.ciudad;
```

En el siguiente ejemplo (E17.IDE) se presenta un programa C que crea una lista de empleados en un arreglo, lee y procesa sus datos para luego imprimir la lista.

🖫 Ejemplo 1.7

```
// E17.CPP
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#define MAX 50
#define SUELDO_HORA 150

typedef struct {
        int dia;
        int mes;
        int anno;
```

```
    } fecha;

typedef struct {
        char cedula[9];
        char nombre[40];
        fecha fecha_pago;
        int horas_trabajadas;
        float sueldo;
} empleado;

// PROTOTIPOS
char menu(void);
int procesa_lista(empleado lista_empleado[MAX]);
void imprime_lista(int numero_emple,empleado lista_empleado[MAX]);

void main() {
        empleado LE[MAX];
        char opcion;
        int num_emp;
        do {
                clrscr();
                opcion=menu();
                clrscr();
                switch(opcion) {
                case '1': clrscr();
                                        num_emp=procesa_lista(LE);
                                        break;
                case '2': clrscr();
                                        imprime_lista(num_emp,LE);
                                        getche();
                                        break;
                }
        } while(opcion!='3');
}

char menu() {
        char opcion;
        printf("[1] Procesar lista de empleados\n");
        printf("[2] Imprimir lista de empleados\n");
        printf("[3] Salir\n");
        opcion= getche();
```

```
                return opcion;
}

int procesa_lista(empleado lista_empleado[MAX]) {
        clrscr();
        int i=0;
        do {
          printf("\nCédula -1 para salir: ");
          scanf("%s",lista_empleado[i].cedula);
          if(strcmp(lista_empleado[i].cedula,"-1")!=0) {
                        printf("Nombre: ");
                        scanf("%s",lista_empleado[i].nombre);
                        printf("Día: ");
                        scanf("%d",&lista_empleado[i].fecha_pago.dia);
                        printf("Mes: ");
                        scanf("%d",&lista_empleado[i].fecha_pago.mes);
                        printf("Año: ");
                        scanf("%d",&lista_empleado[i].fecha_pago.anno);
                        printf("Horas trabajadas: ");
                        scanf("%d",&lista_empleado[i].horas_trabajadas);
                                        lista_empleado[i].sueldo=
(lista_empleado[i].horas_trabajadas)*SUELDO_HORA;
                        i++;
          }
        } while((strcmp(lista_empleado[i].cedula,"-1")!=0) && (i<MAX));
        return i-1;
}

void imprime_lista(int numero_emple,empleado lista_empleado[MAX]) {
  int i;
  clrscr();
  for(i=0;i<=numero_emple;i++) {
            printf("\nCédula: %s\n",lista_empleado[i].cedula);
            printf("Nombre: %s\n",lista_empleado[i].nombre);
            printf("Día: %d\n",lista_empleado[i].fecha_pago.dia);
            printf("Mes: %d\n",lista_empleado[i].fecha_pago.mes);
            printf("Año: %d\n",lista_empleado[i].fecha_pago.anno);
            printf("Horas trabajadas: %d\n",lista_empleado[i].horas_trabajadas);
            printf("Sueldo: %f\n",lista_empleado[i].sueldo);
  }
}
```

Para declarar una hilera o un arreglo en C se puede hacer la asignación de memoria en forma dinámica. Por ejemplo se puede escribir int *x; en lugar de int[10]; y luego reservar memoria en forma dinámica con la función de biblioteca malloc, de la siguiente forma:

```
x = (int *)  malloc(10 * size(int));
```

La sintaxis del malloc() es la siguiente:

```
void *malloc(unsigned número_de_bytes);
```

Se puede mejorar el programa si se lee antes de la asignación de memoria el tamaño exacto del arreglo, por ejemplo analice el siguiente fragmento de programa:

```
int  n, *x;
printf("Cuántos enteros de van a introducir?");
scanf("%d", &n);
x = (int *)  malloc(n * size(int));
```

Usualmente antes de asignar la memoria a la variable se hace una conversión de tipo, por ejemplo, en la sentencia x = (int *) malloc(10 * size(int)); la expresión (int *) tiene este objetivo.

Para liberar la memoria reservada por malloc(), C posee la función de biblioteca free() cuya sintaxis es:

```
void  free(void *);
```

Por ejemplo, para liberar la memoria de un arreglo que fue asignada con malloc():

```
int  n, *x;
printf("Cuántos enteros de van a introducir?");
scanf("%d", &n);
x = (int *)  malloc(n * size(int));
.....
```

```
free(x);
```

La asignación y liberación dinámica de memoria se puede hacer de manera más simple en C++ con los nuevos operadores new y delete (recuerde que estos funcionan si y solamente si se esta compilando en modo C++). La sintaxis es la siguiente:

```
variable_puntero = new tipo_variable;
delete  variable_puntero;
```

Por ejemplo:

```
int *pv;
pv = new int;     // reserva memoria para un entero
if(!pv)  {        // verifica el éxito en la asignación
          printf("Error en la asignación");
          return 1;
}
.............
delete pv;
```

Los operadores new y delete se pueden usar también en forma fácil para asignar y liberar memoria de arreglos, la sintaxis es la siguiente:

```
variable_puntero = new tipo_variable[tamaño];
delete  [tamaño] variable_puntero;
```

Por ejemplo, considere el siguiente fragmento de programa:

```
float *pv;
pv = new float[10]        // reserva memoria para un entero
if(!pv)  {                // verifica el éxito en la asignación
          printf("Error en la asignación");
          return 1;
}
.............
delete [10] pv;
```

Los operadores new y delete son mucho más poderosos que malloc y free, pues permiten asignar y liberar memoria a clases de objetos definidas por el usuario, además pueden ser "sobrecargados".

Tipos definidos por el programador

C permite la existencia de tipos *definidos por el programador* mediante la palabra reservada typedef, la sintaxis es la siguiente:

```
typedef  tipo  nombre_nuevo_tipo;
```

Los tipos definidos por el programador son particularmente útiles cuando se definen estructuras, dado que eliminan la necesidad de escribir repetidamente struct marca en cualquier lugar que una estructura sea referenciada.

Por ejemplo:

```
typedef  struct  {
        char cedula[9];
        char nombre[40];
        int ciudad;
} empleado;

empleado  empleado1, empleado2; // declaración de variables
```

La sintaxis general es la siguiente:

```
typedef  struct  {
        tipo miembro1;
        tipo miembro2;
        ...................
        tipo miembroN;

} nombre_nuevo_tipo;
```

Resulta de suma importancia analizar la relación existente entre punteros y estructuras, pues C posee un operador especial usado cuando

se tienen punteros a estructuras, este operador es -> que se ilustra con el siguiente ejemplo:

```
typedef  struct  {
          char cedula[9];
          char nombre[40];
          int ciudad;
} empleado;

empleado  *p_emple;    // declaración de variables
empleado   emple;

.....................
p_emple = &emple;       // Asignación
```

entonces son equivalentes:

```
emple.cedula      p_emple->cedula      (*p_emple).cedula
```

Para terminar esta sección diremos que la estructura y los punteros son utilizados en C para *estructuras autoreferenciadas*, tales como listas enlazadas y árboles. En el siguiente ejemplo E18.IDE se presenta una lista enlazada lineal de empleados en la que se pueden introducir datos, mostrar la lista y destruirla.

🖫 Ejemplo 1.8

```
// E18.CPP
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>

#define SUELDO_HORA 150

// Lista dinámica de empleados
struct lista_empleados {
          char cedula[10];
          char nombre[40];
          int horas_trabajadas;
```

```
            float sueldo;
        struct lista_empleados *sig;
};

typedef struct lista_empleados nodo;

// Prototipos
char menu(void);
nodo *insertar(nodo *pnodo);
void mostrar(nodo *pnodo);
nodo *destruir(nodo *pnodo);

// Programa princiapal
void main() {
  nodo *primero;
  primero=NULL;
  char opcion;
   do {
     clrscr();
     opcion=menu();
     clrscr();
     switch(opcion) {
          case '1': clrscr();
                    primero=insertar(primero);
                    break;
          case '2': clrscr();
                    mostrar(primero);
                    getche();
                    break;
          case '3': clrscr();
                    primero=destruir(primero);
                    break;
          case '4': clrscr();
                    break;
          default: printf("ERROR al escoger la opción");
     }
   } while(opcion!='4');
}

// Código del menú
char menu() {
```

```
        char opcion;
    printf("[1] Insertar en la lista\n");
    printf("[2] Mostrar la lista\n");
    printf("[3] Destruir la lista\n");
    printf("[4] Salir\n");
    opcion= getche();
    return opcion;
}

// Inserta un empleado a la lista
nodo *insertar(nodo *pnodo) {  // Inserta al inicio de la lista
   nodo *tempo;
   tempo = new nodo;
   printf("Cédula=");
   scanf("%s",tempo->cedula);
   printf("Nombre=");
   scanf("%s",tempo->nombre);
   printf("Horas trabajadas=");
   scanf("%d",&tempo->horas_trabajadas);
   tempo->sueldo=tempo->horas_trabajadas*SUELDO_HORA;
   tempo->sig=pnodo;
   pnodo=tempo;
   return pnodo;
}

// Muestra todos los empleados de la lista
void mostrar(nodo *pnodo) {
  nodo *actual=pnodo;
  while(actual!=NULL) {
     printf("\nCédula: %s\n",actual->cedula);
     printf("Nombre= %s\n",actual->nombre);
     printf("Horas trabajadas= %d\n",actual->horas_trabajadas);
     printf("Sueldo %f\n",actual->sueldo);
     actual=actual->sig;
  }
}

// Destruye o borra toda la lista
nodo *destruir(nodo *pnodo) {
   while(pnodo!=NULL) {
      nodo *tempo=pnodo;
```

```
        pnodo=pnodo->sig;
    delete(tempo);
  }
  return NULL;
}
```

1.7 Archivos

Cuando se trabaja con archivos secuenciales de datos, el primer paso es establecer el *área de buffer*, donde la información se almacena temporalmente mientras se está transfiriendo entre la memoria del computador y el archivo de datos en el disco duro o diskette. Esto permite que el proceso de escritura y lectura de información sea mucho más rápido. Para establecer esta área de buffer se escribe la sentencia:

```
FILE  *p_archivo;
```

donde FILE (debe estar escrito en mayúscula) es un tipo especial de estructura definido en la biblioteca stdio.h y p_archivo se conoce como el puntero al archivo secuencial.

Para abrir el archivo se usa la función de biblioteca fopen, cuya sintaxis es la siguiente:

```
p_archivo = fopen(nombre_archivo, tipo_archivo);
```

donde nombre_archivo será una hilera que almacena el nombre con que se grabará el archivo y debe seguir las reglas usadas por el sistema operativo (en nuestro caso DOS), y tipo_archivo es una hilera escogida de la siguiente tabla:

Tipo de archivo	Significado
"r"	Abrir archivo para leer
"w"	Crear un archivo para escribir
"a"	Añadir en el archivo
"rb"	Abrir archivo binario para leer
"wb"	Crear un archivo binario para escribir
"ab"	Añadir en el archivo binario
"r+b"	Abrir archivo binario para leer/escribir
"w+b"	Crear un archivo binario para leer/escribir
"a+b"	Añadir en el archivo binario o para leer/escribir
"a"	Añadir en el archivo
"rt"	Abrir archivo de texto para leer
"wt"	Crear un archivo de texto para escribir
"at"	Añadir en el de texto binario
"r+t"	Abrir archivo de texto para leer/escribir
"w+t"	Crear un archivo de texto para leer/escribir
"a+t"	Añadir en el archivo de texto o para leer/escribir

Lafunc

ión fopen retorna un puntero que apunta al área de buffer asociada al archivo, además

retorna NULL si no se puede abrir el archivo.

Para cerrar un archivo se usa la función de biblioteca fclose, cuya sintaxis es la siguiente:

```
fclose(p_archivo);
```

Típicamente un programa con archivos tiene al menos las siguientes sentencias o instrucciones:

```
#include <stdio.h>
  FILE *archivo;
  if((archivo = fopen("DATOS.BIN", "wb")) == NULL)
     printf("ERROR en la creación del archivo");
   .....................
  fclose(archivo);
```

C posee una amplia gama de tipos y formas de manipular archivos, presentaremos únicamente los archivos binarios del sistema ANSI que son muy utilizados.

Las principales funciones de biblioteca en este sistema de archivos son las siguientes:

```
unsigned fread(void *variable, int numero_bytes,int contador,FILE *archivo);

unsigned fwrite(void *variable, int numero_bytes,int contador,FILE *archivo);

int fseek(FILE *archivo, long numero_bytes, int origen);
```

donde origen puede ser:

Origen	Nombre del macro
Principio del archivo	SEEK_SET
Posición actual	SEEK_CUR
Fin Archivo	SEEK_END

Las funciones fread y fwrite pertenecen a la biblioteca STDIO.H, los parámetros de estas se describen en la siguiente tabla:

Parámetro	Finalidad
variable	Puntero al bloque de memoria RAM donde se almacena la información.
numero_bytes	Longitud en bytes de cada ítem que van a ser leído.
contador	Número de ítems (bloques) que van a ser leídos.
archivo	Puntero al área de buffer (stream).

En caso de éxito en la lectura, fread retorna el número de ítems (no de bytes) leídos, en caso de error o bien al final del archivo, fread retorna 0 .

En las funciones fwrite y fread los parámetros tienen exactamente las mismas finalidades que los de la función fread.

Estas funciones se ilustran en el ejemplo E19.IDE, en el que se presentan varias funciones ilustrativas, entre ellas una para insertar datos de un empleado en un archivo, otra para mostrar todo los registros de un archivo y otra para buscar un empleado (registro) en un archivo.

🖫 Ejemplo 1.9. Lectura y escritura en un archivo binario

```
// E19.CPP
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>

#define SUELDO_HORA 150
```

```
// Define un tipo booleano
enum boolean { false=0, true };

// Estructura de registro para el empleado
struct empleado {
        char cedula[10];
        char nombre[40];
        int horas_trabajadas;
        float sueldo;
};

typedef struct empleado nodo;
// Prototipos
char menu(void);
void desplegar_empleado(nodo emple);
boolean insertar_archivo(void);
boolean buscar_empleado(char cedula[10]);
boolean mostrar_archivo(void);

// Programa principal
void main() {
  char ced[10];
  boolean resul;
  char opcion;
   do {
    clrscr();
    opcion=menu();
    clrscr();
    switch(opcion) {
        case '1': clrscr();
                  resul=insertar_archivo();
                  if(resul==false) {
                    clrscr();
                    printf("ERROR");
                    getche();
                  }
                  break;
        case '2': clrscr();
                  resul=mostrar_archivo();
                  if(resul==false) {
```

```
                        clrscr();
                      printf("ERROR");
                      getche();
                    }
                      break;
          case '3': clrscr();
                      printf("Digite la cédula: ");
                      scanf("%s",ced);
                      resul=buscar_empleado(ced);
                      if(resul==false) {
                        clrscr();
                        printf("No se encuentra en el archivo");
                        getche();
                      }

                      break;
          case '4': clrscr();
                      break;
          default: printf("ERROR al escoger la opción");
      }
    } while(opcion!='4');
}

// Código del menú
char menu() {
    char opcion;
    printf("[1] Insertar en el archivo\n");
    printf("[2] Mostrar el archivo\n");
    printf("[3] Buscar en el archivo\n");
    printf("[4] Salir\n");
    opcion= getche();
    return opcion;
}

// Despliega un empleado en pantalla
void desplegar_empleado(nodo emple) {
  printf("Cédula= %s\n",emple.cedula);
  printf("Nombre= %s\n",emple.nombre);
  printf("Horas trabajadas=: %d\n",emple.horas_trabajadas);
  printf("Sueldo=: %f\n",emple.sueldo);
}
```

```
// Inserta un empleado en el archivo
boolean insertar_archivo() {
   nodo emp_tempo;
   FILE *archivo;
   if((archivo = fopen("DATOS.BIN", "wb")) == NULL) {
      return false;
   }
   do {
     clrscr();
     printf("Cédula (-1 para salir)= ");
     scanf("%s",emp_tempo.cedula);
     if(strcmp(emp_tempo.cedula,"-1")!=0) {
           printf("Nombre=");
        scanf("%s",emp_tempo.nombre);
        printf("Horas trabajadas=");
        scanf("%d",&emp_tempo.horas_trabajadas);
        emp_tempo.sueldo=emp_tempo.horas_trabajadas*SUELDO_HORA;
        fwrite(&emp_tempo,sizeof(nodo),1,archivo);
     }
   } while(strcmp(emp_tempo.cedula,"-1")!=0);
   fclose(archivo);
   return true;
}

// Busca un empleado por número de cédula
boolean buscar_empleado(char cedula[10]) {
  FILE *archivo;
  nodo emple_tempo;
  boolean encontro=false;
  if((archivo = fopen("DATOS.BIN", "rb")) == NULL)   {
     return false;
  }
  do {
    fread(&emple_tempo,sizeof(nodo),1,archivo);
    clrscr();
    if(strcmp(emple_tempo.cedula,cedula)==0) {
        desplegar_empleado(emple_tempo);
            getche();
        encontro=true;
    }
```

```
      } while((!feof(archivo)) && (encontro==false));
  fclose(archivo);
  return true;
}

// Muestra el archivo completo
boolean mostrar_archivo(void) {
  FILE *archivo;
  nodo emple_tempo;
  if((archivo = fopen("DATOS.BIN", "rb")) == NULL)   {
     return false;
  }
  do {
    fread(&emple_tempo,sizeof(nodo),1,archivo);
    clrscr();
    desplegar_empleado(emple_tempo);
    getche();
  } while(!feof(archivo));
  fclose(archivo);
  return true;
}
```