Lenguaje C

Organización de un programa en C

Todo programa fuente, y en particular aquellos escritos en lenguaje C, obedecen a una cierta organización, sin la cual el compilador no podría cumplir con su objetivo. Podemos identificar los siguientes elementos básicos en un programa fuente:

1. **Directivas de preprocesado**. Es una instrucción para que el compilador tome alguna acción determinada. Cuando se utiliza, el primer carácter de la línea que lo contiene debe ser un #.

2. **Declaración de Variables**. Una variable es un elemento que asocia un nombre a un dato. El nombre de la variable es un símbolo que sirve para representar su valor.

3. **Expresiones**. Una expresión es una secuencia de operadores, operando y separadores que debe ser evaluada. El resultado de esta evaluación puede ser asignado a una variable, servir como elemento de decisión para el flujo del programa, o bien puede ser empleado como resultado parcial en otra expresión que contiene a la expresión original.

4. **Comentarios**. Puesto que en general es deseable mantener y perfeccionar las aplicaciones en el tiempo, debe existir alguna forma expedita de explicar y/o recordar el funcionamiento del programa a quienes se encargan de su mantención y crecimiento. Ello se hace mediante comentarios que se insertan en el código fuente.

   Para diferenciar el código fuente de los comentarios, de modo que a estos últimos no los considere el compilador, se los distingue mediante separadores.

```
/* Indica comentarios que abarcarán
   varias líneas dentro del código fuente. */

// Indica comentarios de una sola línea.
```

Tipos de operadores

Los operadores pueden clasificarse en aritméticos, relacionales y lógicos. También pueden clasificarse en unitarias o binarias en caso de que requieran uno, dos o más operandos para realizar la operación.

- **Operadores aritméticos**. Permiten realizar operaciones aritméticas tradicionales como sumas o multiplicaciones.
- **Operadores lógicos**. Realizan evaluaciones para evaluar la veracidad de una proposición a nivel de bits.
- **Operadores relacionales**. Permiten la comparación entre diferentes datos; siempre deben compararse datos de un mismo tipo.
- **Otros operadores**. Posibilitan el trabajo de diferentes acciones dentro del lenguaje, como el manejo dinámico de memoria, la decisión, la selección, o los incrementos.

Los operadores más comunes se describen a continuación.

Operador	Nombre	Uso
&	Dirección	Manejo de memoria
*	Indirección	Manejo de memoria
++	Incremento	Incrementos
--	Decremento	Incrementos
+	Suma	Operaciones aritméticas
–	Resta	Operaciones aritméticas
*	Multiplicación	Operaciones aritméticas
/	División	Operaciones aritméticas
%	Módulo	Operaciones aritméticas
<<	Corrimiento izquierdo	Operaciones a nivel de bits
>>	Corrimiento derecho	Operaciones a nivel de bits
&&	AND	Operaciones lógicas
||	OR	Operaciones lógicas
!	Negación	Operaciones lógicas
=	Asignación	Asignación de valores
<	Menor que	Relaciones
>	Mayor que	Relaciones
<=	Menor o igual que	Relaciones
>=	Mayor o igual que	Relaciones
==	Igual	Relaciones
!=	Diferente	Relaciones
.	Selector de componente	Manejo de estructuras
X ? Y : Z	Condicional	Decisión `If X then Y; else Z`

Directivas de preprocesado

El preprocesado es una fase anterior a la compilación de un programa del lenguaje C; se aplica sobre el archivo principal que contiene el código fuente. Su función principal es hacer comprensible el código fuente para el compilador, cambiando las llamadas directivas de preprocesamiento. Estas directivas siempre están señaladas por un carácter especial para que solo sean modificadas por el preprocesador, en el caso del preprocesador de C el símbolo utilizado es #. Otra parte muy importante de los preprocesadores, son los macros: funciones muy cortas que reemplazan expresiones completas dentro del código fuente.

Directiva #include

Todo código en C debe comenzar por la inclusión de las bibliotecas necesarias para el funcionamiento correcto del programa. Estas inclusiones se logran mediante la directiva #include, cuya sintaxis es:

```
#include <biblioteca.h>
```

El utilizar los símbolos <, > indica una inclusión desde la carpeta de bibliotecas del compilador. Si se desea añadir un archivo .h desde otra ruta (como puede ser las bibliotecas creadas por el programador) se utiliza la siguiente sintaxis:

```
#include "ruta_de_la_biblioteca/biblioteca.h"
```

La inclusión de bibliotecas permite utilizar funciones creadas con anterioridad o que son necesarias para la correcta interacción entre la computadora y el usuario.

Directiva #define

El uso de esta directiva permite la introducción de las constantes simbólicas. Una constante de este tipo es un nombre que sustituye una secuencia de caracteres; estos caracteres pueden representar una constante numérica, alfabética o una cadena. Cuando un programa es compilado, cada constante simbólica es sustituida por la secuencia de caracteres correspondiente. La sintaxis de la directiva #define es:

```
#define nombre cadena_caracteres
```

Tipos de datos

En C existen tipos de datos, los cuales contribuyen a la diversidad de información que se puede manejar; cada uno puede ser representado de forma diferente en la memoria de la computadora. La forma de declarar una variable es:

```
tipo_dato nombre |= valor|;
```

El campo valor es opcional y permite iniciar una variable en tiempo de compilación; es decir, la variable tendrá un dato fijo al momento de comenzar la ejecución del programa.

Los tipos de datos básicos en C son:

Tipo	*Descripción*	*Tamaño*	*Rango*
int	Número entero,	2 Bytes	-32768 a 32767
char	Carácter.	1 Bytes	-128 a 127
float	Número en punto flotante (fracción).	4 Bytes	3.4exp(-38) a 3.4exp(38)
double	Número en punto flotante de doble precisión.	8 Bytes	1.7exp(-308) a 1.7exp(308)
void	Sin tipo de dato.	0 Bytes	Sin valores

Los cualificadores que afectan a los tipos de datos y permiten los tipos combinados son:

Tipo	*Descripción*
unsigned	Número sin signo
long	Carácter.
short	Número en punto flotante (fracción).

Las variables no son las únicas expresiones en C que tienen asignado un tipo de dato, también las funciones lo tienen, y es necesario especificarlo para la especificación de parámetros o la devolución de resultados.

Formaciones

C puede manejar conjuntos de datos con características comunes. Las formaciones son datos de un mismo tipo que comparten un mismo nombre, y cada dato es manejado de manera individual por medio de un índice; si la formación es de tipo carácter, entonces de llama cadena. Las formaciones pueden ser unidimensionales, bidimensionales, etc.; siempre son declaradas como variables, utilizando un componente extra que indica el número de lugares disponibles:

```
tipo_dato nombre[número_posiciones] |= {valores}|;
```

El campo valores es opcional y sirve para iniciar el arreglo en tiempo de compilación. Debe aclararse que los valores deben separarse por comas y ajustarse al número de posiciones del arreglo; es decir, si un arreglo tiene n posiciones, entonces deben existir n valores separados por comas entre las llaves.

Las cadenas de caracteres pueden manejarse de dos formas: toda la formación como un único dato, así como cada elemento dentro del arreglo.

Instrucciones de entrada y salida de datos

Un programa necesita entradas y salidas de datos para procesar la información y mostrar al usuario el resultado del problema que se está resolviendo. Las instrucciones de entrada toman desde teclado los datos que el usuario ingrese en el programa, en tanto que las de salida muestran los resultados directamente en la pantalla de la computadora. Dependiendo del tipo de dato que se esté manejando es recomendable utilizar una u otra instrucción, así como la conversión de datos a la entrada y salida.

Instrucciones de entrada

Las instrucciones de entrada más utilizadas para leer datos desde teclado son:

Función getchar. Lee un carácter individual desde el teclado, asignando a una variable el valor leído. Su sintaxis es:

```
variable = getchar();
```

Función gets. Lee una cadena de caracteres introducida desde teclado; la función leerá la cadena hasta que encuentre un carácter de nueva línea (tecla entrar). Su sintaxis es:

```
gets(cadena);
```

Función scanf. Esta es una función de entrada de datos, que permite la manipulación de varios tipos de datos mediante cadenas de control. Su sintaxis es:

```
scanf("cadena_de_control", argumento1, argumento2,..., argumentoN);
```

La cadena control especifica los tipos de datos de cada argumento, utilizando caracteres de conversión de datos; los argumentos son los nombres de las variables que se están capturando, precedidos cada uno del símbolo &. Los nombres de formaciones no deben llevar el ámpersand.

Instrucciones de salida

Cada instrucción de entrada tiene su contraparte de salida. Estas instrucciones permiten cerrar el flujo de datos desde dentro del programa:

Función putchar. Muestra un carácter individual en el monitor, tomando a una variable como parámetro. Su sintaxis es:

```
putchar(variable);
```

Función puts. Muestra una cadena de caracteres en el monitor; la función escribirá una cadena almacenada en una variable. Su sintaxis es:

```
puts(cadena);
```

Función printf. Esta es una permite la escritura de cadenas de caracteres, así como de varios tipos de datos mediante cadenas de control. Su sintaxis es:

```
printf("cadena_de_control", argumento1, argumento2,..., argumentoN);
```

La cadena control especifica el formato que debe tener la salida de datos, incluyendo mensajes, caracteres de conversión de datos y secuencias de escape; los argumentos son los nombres de las variables que se están mostrando.

Caracteres de conversión y secuencias de escape

Los caracteres de conversión permiten indicar al compilador que tipo de variable se está capturando o imprimiendo con las instrucciones scanf y printf. Estos caracteres siempre van precedidos del símbolo %, para indicar que a continuación se especifica un tipo de dato. Los más comunes son:

Carácter	*Significado*
d	Entero decimal
i	Entero decimal, octal o hexadecimal
o	Entero octal
x	Entero hexadecimal
u	Entero decimal sin signo
c	Carácter
s	Cadena de caracteres
f	Número en punto flotante
[…]	Cadena de caracteres hasta un símbolo en particular

Las secuencias de escape permiten dar formato específico a las cadenas utilizadas en la función printf. Las más socorridas se muestran a continuación:

Secuencia	*Descripción*	*Secuencia*	*Descripción*
\a	Sonido audible	\\	Barra invertida
\b	Retroceso	\'	Apóstrofe
\n	Saltar una línea	\"	Doble comilla
\r	Retorno de carro	\?	Interrogación
\t	Tabulación horizontal	\o	Cadena de dígitos octales
\v	Tabulación vertical	\x	Cadena de dígitos hexadecimales

Instrucciones de control I

Las instrucciones de control son parte esencial de cualquier programa en lenguaje C, ya que implementan las diferentes estructuras básicas de un algoritmo: estructura secuencial, estructura condicional y estructura iterativa.

Instrucción condicional: if – else

Esta instrucción permite establecer una decisión lógica con base en una expresión dada; la expresión que será validad puede ser un conjunto relacionado con operadores lógicos, de manera que se evalúen diferentes casos en una sola decisión. Dentro de la decisión existen dos diferentes caminos para seguir:

- La condición es válida y se ejecuta el bloque de instrucciones debajo de if.
- La condición no es válida y el bloque de instrucciones que se ejecutará está debajo de else.

Cabe destacar que la sección else puede o no existir, dependiendo del diseño del algoritmo. La sintaxis de este tipo es:

```
if(expresión) {
      bloque_instrucciones;
}
```

La sintaxis de la instrucción if – else completa es:

```
if(expresión) {
      bloque_instrucciones;
}
else {
      bloque_instrucciones;
}
```

Si el bloque de instrucciones consiste en una sola línea, las llaves de agrupación pueden omitirse (tanto debajo de if como de else).

Instrucción condicional: switch

Dentro de la programación existen situaciones en las cuales se deben implementar múltiples decisiones. Esto haría más extenso el código fuente, incluso se prestaría a confusión la anidación de instrucciones if con muchos casos por evaluar.

Para evitar los anidamientos innecesarios se establece como función multidecisión a la instrucción switch, la cual evalúa una expresión, y dependiendo del resultado ejecuta el bloque de instrucciones acorde al dictamen de la decisión; si no encuentra algún caso válido, puede ejecutar un bloque por defecto de manera opcional. Su sintaxis es:

```
switch(expresión) {
       case expresión1:
              instrucciones;
              break;
       case expresión2:
              instrucciones;
              break;
          ⋮
       case expresiónM:
              instrucciones;
              break;
       default:
              instrucciones;
              break;
```

```
}
```

Operador condicional

Este operador permite establecer una condición if – else sencilla y rápida.

```
expresión1 ? expresión2 : expresión3;
```

Si la expresión 1 es cierta, entonces se evalúa la expresión 2; de lo contrario se evalúa la expresión 3. Finalmente, el valor de la expresión condicional será el de la expresión que es evaluada.

Dependiendo del uso que se le quiera dar o que se requiera, se puede optar por cualquiera de las tres expresiones condicionales existentes. El único punto que debe considerarse es la complejidad del problema a resolver; dependiendo del caso, es posible que algunos if anidados sean más convenientes que un switch, o un operador condicional más que el if. Este punto recalca su importancia en el tiempo de procesado de las instrucciones.

Instrucciones de control II

Las instrucciones de control iterativas permiten la repetición de tareas sin necesidad de reescribir las mismas instrucciones consecutivamente.

Instrucción iterativa: while

Esta instrucción ejecuta repetidamente un grupo de instrucciones, hasta que se cumpla una determinada condición. La condición es validada al inicio de cada bucle; por lo tanto, es de esperarse que el ciclo pueda o no ejecutarse, ya que si la condición se satisface desde un inicio, el conjunto de instrucciones dentro de while no se ejecutará. Su sintaxis general es:

```
while(expresión) {
      bloque_instrucciones;
}
```

La expresión necesita alguna modificación dentro del bloque de instrucciones, proporcionándole al bucle una salida; en caso contrario, las instrucciones serán ejecutadas infinitamente.

Instrucción iterativa: do – while

Habrá ocasiones, en las cuales la verificación de la condición de parada de un ciclo necesita realizarse al final de cada pasada; aquí es conveniente utilizar la instrucción do – while. Trabaja exactamente igual que while, excepto el lugar de validación de la expresión. Su sintaxis es:

```
do{
      bloque_instrucciones;
} while(expresión);
```

Instrucción iterativa: for

La instrucción iterativa for implementa bucles donde el número de pasadas se 'conoce' con anticipación. Requiere tres expresiones: una de inicio del ciclo, una que determine si se ejecuta o no el bucle, y una última que modifica la condición de paro al final de cada pasada; las tres expresiones manipulan un parámetro llamado índice de ciclo. Su sintaxis es:

```
for(expresión1; expresión2; expresión3) {
      bloque_instrucciones;
}
```

Rompiendo un ciclo

Un ciclo se rompe con las instrucciones break (para salir completamente del ciclo) y continue (para terminar la pasada actual y ejecutar la siguiente). Sus sintaxis son sencillas:

```
break;

continue;
```

Funciones

Una función es un segmento de programa que realiza funciones bien definidas. La función procesará la información que le es pasada desde el punto del programa en donde se accede a ella y devolverá un solo valor. Este es el claro ejemplo de la programación modular en lenguaje C: un problema puede dividirse en problemas más pequeños que se resolverán independientemente; al final se conjunta la solución general.

Declaración de funciones

La definición de una función tiene dos componentes esenciales: la primera línea y el cuerpo de la función. En la primera línea se establecen el tipo de valor devuelto por la función, seguido del nombre de la función y (opcionalmente) un conjunto de argumentos.

```
tipo_dato nombre_funcion(tipo arg1, tipo arg2, tipo arg3,… ,tipo argN) {
      instrucciones;
      return expresion;
}
```

La instrucción return permite devolver un valor desde la función hasta el punto donde se le llamó; también permite que se devuelva el control del programa hasta el punto de la llamada.

Llamada de una función

Se puede acceder a una función especificando su nombre, seguido de la lista de argumentos necesitados para que pueda accionar. La llamada a la función puede formar parte de una expresión simple o puede ser un operando de una expresión más compleja. Los argumentos de una función pueden ser constantes, variables simples, o expresiones más complejas como arreglos o apuntadores.

Las llamadas a una función pueden presentarse de la siguiente manera:

```
y = polinomio(x);       /*Función que devuelve un valor, el cual se asigna a la variable
                        y.*/
suma(a, b, c);          /*Función que no devuelve algún valor, y que pide los parámetros
                        a, b y c.*/
imprime();              /*Función que no devuelve ni pide valor alguno.*/
```

Prototipos de función

Es la declaración de la primera línea de una función antes de la función principal, y alerta al compilador que después del main se definirá la función a la que hace alusión el prototipo. La declaración de estas expresiones siempre será antes del main y después de las instrucciones de preprocesado. Es importante aclarar que se trata únicamente de una declaración (como en una variable) y por lo tanto, debe terminar con punto y coma:

```
tipo_dato nombre_funcion(tipo arg1, tipo arg2,… ,tipo argN);
```

Paso de argumentos a una función

Cuando se le pasa un valor simple a una función, se copia el valor al argumento de la función; ahí trabaja con dicho argumento durante todo el proceso. Para que se pueda completar la llamada de funciones es necesario pasar el mismo número de valores que se especifican en la declaración de la función.

Variables globales y locales

Existen dos tipos de variables que permiten manejar los datos de una u otra manera; éstas son conocidas como variables locales o variables globales. Cada una de ellas permitirá manejo interno de datos dentro de una función específica, o bien, una manipulación en todo el programa. Las variables locales se declaran dentro de una función específica:

```
int IDEA(char *A, char *B) {
      int i, j, indice;        //Variables locales
      bloque_de_instrucciones;
}
```

Las variables globales deben declararse fuera de cualquier función que componga al programa; así es posible manipularlas desde cualquier punto en cualquier momento. Obviamente, la alteración en el dato de la variable afectará al resultado de cualquier función que trabaje con él.

```
#include <stdio.h>
#define RONDA 8

char *Mcla, *Crip, *K;         //Variables globales

int IDEA(char *A, char *B);

main(){
      bloque_de_instrucciones;
}
```

Recursividad

La recurrencia o recursividad es la forma en la cual se especifica un proceso basado en su propia definición; es decir, un proceso se realiza invocándose a sí mismo un número determinado de veces. La dificultad de la recurrencia implica que el algoritmo debe ser lo suficientemente exacto para parar en un determinado número de veces, o las llamadas sucesivas tenderán al infinito. La recursividad se presenta clásicamente al programar el factorial de un número; en la

Matemática esta característica se manifiesta en la geometría fractal, que se considera una aplicación sucesiva de la composición de una transformación lineal consigo misma.

La máquina de Turing estudia los posibles casos de recursividad en los lenguajes formales.

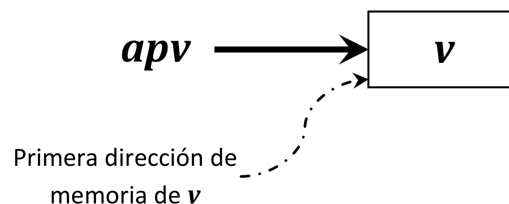
Apuntadores

Cada dato que se almacena en la memoria de la computadora ocupa una o varias celdas contiguas; el número de celdas necesitadas depende del dato que se requiera guardar. En ocasiones es necesario conocer la dirección de memoria donde se almacena dicha variable, en lugar del dato que contiene. Para ello nace la necesidad de utilizar apuntadores.

Un puntero o apuntador es una variable que hace referencia a una región de memoria; es una variable cuyo valor es una dirección de memoria. Trabajar con apuntadores implica el manejo de la memoria donde se almacenan las variables.

Al declarar una variable *v* (de cualquier tipo de dato), el compilador la almacena automáticamente en alguna de las localidades de memoria asignadas al usuario. El dato que representa *v* se puede acceder al conocer la localización (dirección) de la primera celda de memoria asignada; dicha dirección se determina mediante *&v*, donde *&* es el conocido operador dirección.

La dirección de *v* puede asignarse a otra variable, por ejemplo *apv*; entonces se tendría que la variable *apv = &v*. La nueva variable es conocida como apuntador a *v*, ya que señala la dirección de memoria donde ésta última se almacena. Se debe recalcar que *apv* representa la dirección de *v*, no su valor. Así *apv* es una variable de tipo apuntador al tipo de dato de *v*.



Mediante la expresión **apv* se puede acceder al dato almacenado en *v*, donde *** es el operador indirección, que únicamente maneja variables apuntadoras.

La declaración y uso de apuntadores debe ser preciso. Al declarar una variable apuntadora es necesario que coincida con el tipo de dato de la variable que será apuntada.

```
int v;      // Variable de tipo entero
int *apv;   // Apuntador a una variable de tipo entero
apv = &v;   // El apuntador señala a la primera celda de memoria de v.
u = *apv;   // El valor de v es asignado mediante su apuntador a u.
```

Es importante señalar que las formaciones (arreglos) son realmente apuntadores; es decir, al declarar un arreglo se declara un apuntador que señala a la primera dirección de memoria de un bloque de varios datos del mismo tipo. Las siguientes declaraciones son equivalentes:

```
int A[20];
int *A;
```

La principal diferencia radica en el número de celdas que se reservaran; mientras que el arreglo ya tiene definido el espacio que ocupará, el apuntador puede variar su longitud durante la ejecución del programa.

Manejo dinámico de la memoria

Cada vez que se ejecuta un programa es necesario asignar una cantidad de la memoria para los datos que manejará. Este uso de memoria se denomina manejo de la memoria, y se divide en: manejo estático, manejo dinámico y manejo automático.

Asignación estática. Consiste en asignar memoria en tiempo de compilación (antes de que el programa asociado sea ejecutado). La asignación estática ya es fija es no puede cambiar durante la ejecución del programa, a diferencia de la asignación dinámica o la automática donde la memoria se asigna a medida que se necesita en tiempo de ejecución.

Asignación automática. Las variables automáticas son variables locales a un bloque de sentencias (subrutina, función o procedimiento). Pueden ser asignadas automáticamente en la pila de datos cuando se entra en el bloque de código; cuando se sale del bloque, las variables son automáticamente liberadas. Las variables automáticas tendrán un valor sin definir cuando son declaradas.

Asignación dinámica. Es la asignación durante el tiempo de ejecución de un programa. Es una manera de distribuir la propiedad de recursos de memoria limitada entre muchas piezas de código y datos. Una variable asignada dinámicamente permanece asignado hasta que es liberada explícitamente.

Para realizar esta asignación, C cuenta con dos funciones importantes de la biblioteca stdlib.h:

```
int *X;
X = (int *) malloc(20 * sizeof(int));
free(X);
```

La primera función malloc (*memory allocation*) permite reservar una cantidad específica de la memoria, atendiendo a un valor determinado (en este caso 20 veces el tamaño de una variable entera); mientras tanto, la segunda función free permite liberar esa memoria reservada con malloc una vez que se ha utilizado.

Las partes que integran la asignación de la memoria son:

- Conversión de datos (type casting). Quiere decir que un tipo de dato evolucionará en otro.
- Reserva de memoria. La variable se destina a un bloque dinámico de memoria.
- Tamaño del bloque. Indica que el bloque dinámico tendrá una longitud para varios datos del mismo tipo.

Manejo de archivos de datos

Cuando se trabaja con archivos secuenciales, el primer paso es establecer un área de búfer, donde la información se almacena temporalmente mientras se está transfiriendo entre la memoria de la computadora y el archivo de datos. Para establecer que un programa trabajará con archivos es necesario incluir en la declaración de variables la declaración de un puntero tipo archivo:

```
FILE *archivo;
```

Un archivo siempre debe abrirse antes de comenzar a utilizarse. Esto asocia el nombre del archivo con el búfer de datos reservado; también se especifica con qué tipo de archivo se trabajará: sólo lectura, sólo escritura, o lectura/escritura. Un archivo se abre con la instrucción

```
archivo = fopen("ruta/nombre_archivo.ext", "tipo_archivo");
```

Donde ruta/nombre_archivo.ext es una cadena de texto que especifica el nombre y la ubicación del archivo que se utilizará; y tipo_archivo corresponde a otra cadena de texto para indicar cómo se trabajará con él. La siguiente tabla muestra los tipos de archivo posibles:

Tipo	Acción
"r"	Abre un archivo existente para sólo lectura.
"w"	Crea un archivo para escritura. Si el archivo ya existe, será sobrescrito.
"a"	Abre un archivo existente para añadir nueva información al final. Se crea uno nuevo si no existe el archivo especificado.
"r+"	Abre un archivo existente para lectura y escritura.
"w+"	Crea un archivo para lectura y escritura. Si el archivo ya existe, será sobrescrito.
"a+"	Abre un archivo existente para leer y añadir información al final. Se crea uno nuevo si no existe el archivo especificado.

Cuando se deja de trabajar con un archivo, es necesario cerrarlo y liberar el espacio de memoria reservado para el manejo de sus datos. Esto se hace con la instrucción

```
fclose(archivo);
```

Para procesar archivos existen las funciones de biblioteca fprintf (permite escribir información en un archivo) y fscanf (permite leer información desde un archivo), dos funciones análogas a las conocidas prinf y scanf. Sus sintaxis son:

```
fprintf(archivo, "cadena_control", argumento1, argumento2,…, argumentoN);
fscanf(archivo, "cadena_control", argumento1, argumento2,…, argumentoN);
```

Estructuras

Las estructuras son formaciones de datos, que poseen elementos de diferentes tipos de datos; es decir, una estructura puede tener números enteros, flotantes, caracteres, etc. y manejarlos de forma individual.

A diferencia de las formaciones, las estructuras se declaran de una manera mucho más compleja, ya que cada miembro debe definirse explícitamente. La sintaxis de este tipo de declaraciones es:

```
struct marca {
     tipo_dato1 miembro1;
     tipo_dato2 miembro2;
     ⋮
     tipo_datoN miembroN;
} variable1, variable2, ..., variableM;
```

Cada parte de la sintaxis se resume así:

- Marca. El nombre de estructura; así se conocerá a todas las estructuras dentro del programa.
- Miembro. Son los datos que se engloban dentro de la estructura, y que pueden manejarse de forma individual o en conjunto.
- Variable. Es el nombre de una estructura en particular que pertenece al tipo Marca; es decir, es el identificador de una estructura que se manejará en el programa.

Una estructura también puede definir un nuevo tipo de dato; es decir, en lugar de manejar tipos de datos convencionales por separado, se pueden definir nuevos tipos de datos con base en una estructura. Para definir estas variables se utiliza la sintaxis:

```
typedef struct marca {
     tipo_dato1 miembro1;
     tipo_dato2 miembro2;
     ⋮
     tipo_datoN miembroN;
};
```

En este caso, dentro del programa principal se pueden definir variables del tipo que enuncia la marca en la sintaxis anterior:

```
main() {
     int x, y;
     marca variable1, variable2,… variableN;
}
```

Independientemente de la manera en que se define una estructura (como estructura o como nuevo tipo de dato) cada uno de estas variables se maneja con base en sus miembros individuales o en conjunto, así como se utilizan los arreglos o los apuntadores. La diferencia es utilizar el operador de selección (.), el cual permite identificar el miembro que se está procesando durante la ejecución de una instrucción. De esta forma el miembro que se selecciona es el único que sufre modificaciones.

Por ejemplo, se define una estructura:

```
typedef struct ejemplo {
     int x;
     char y;
};
```

Y se utiliza en la función principal:

```
main() {
     ejemplo A, B;
     scanf("%d,%c", &A.x, &A.y);    /*Manejo de cada miembro en forma
                                    individual.*/
     B = A;                         //Manejo de la estructura completa.
}
```

En este ejemplo se observa, cómo se puede manipular la estructura miembro por miembro o en conjunto; aunque se definió un nuevo tipo de dato, este manejo es válido para las estructuras en general.