Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Antología

Materia de Programación I

Profesor

M.C. Yolanda Moyao Martínez

Fecha

Enero 2017

Índice

Introducción

Esta antología tiene como fin ayudar a los estudiantes del curso de programación I durante el aprendizaje de la materia ya que el estudio del lenguaje C, así lo requiere pues es un lenguaje básico para el estudio de los lenguajes de programación.

En primera instancia se tiene una introducción a los lenguajes, para continuar con una explicación de cómo se desarrolla un programa en lenguaje C. Se explican las estructuras utilizadas dentro del lenguaje, iniciando con las estructuras de secuencia, posteriormente con las de decisión, para concluir con los ciclos; en este tema se revisan los arreglos para poder conocer su manejo en general. Un tema importante es el uso de apuntadores, en el que se trabajan funciones del lenguaje, esto servirá para el manejo de funciones y paso de parámetros. Se concluye este trabajo con el manejo de archivos para una funcionalidad del lenguaje.

Además de las funciones de la biblioteca estándar, la mayoría de los compiladores de C también proporcionan otras funciones que no son estándares, pero, éstas suelen ser muy útiles para el programador, quien, por otra parte, también puede desarrollar sus propias funciones.

Fases de la Puesta a Punto de un Programa en Lenguaje C

¿Qué pasos hay que dar para convertir un algoritmo en un programa?

Para convertir un algoritmo en un programa, se deben llevar a cabo las siguientes fases de puesta a punto:

Edición. En esta primera fase, el programador debe convertir el algoritmo que haya diseñado en instrucciones escritas en C. Para ello, se debe hacer uso de un editor de textos, con el cual se obtendrá el llamado código fuente del programa. El programador deberá guardar dicho código fuente en un archivo con extensión (.c) o (.cpp).

Si un programa escrito en C se va a compilar con un compilador de C, el código fuente debe ser guardado, obligatoriamente, en un archivo con extensión (.c). Ahora bien, si se utiliza un compilador de C/C++, el archivo se puede guardar con extensión (.c) o (.cpp).

Preproceso. El preproceso sirve para realizar modificaciones en el código fuente obtenido en la fase de edición. Es el programador quien, mediante directivas del preprocesador, "dice" al preprocesador las modificaciones que éste debe llevar a cabo.

El preprocesador es un programa característico de C, es decir, en otros lenguajes de programaciónno existe, y siempre se ejecuta antes de llevarse a cabo la compilación. Esto es debido a que, es el propio compilador quien llama al preprocesador antes de realizar la compilación.

Compilación. Una vez que el código fuente ha sido preprocesado, el compilador traducirá ese código fuente (modificado) a código máquina, también llamado código objeto, siempre y cuando, el propio compilador no detecte ningún error en dicho código fuente ya preprocesado.

Como resultado de la compilación, el compilador guardará el código objeto del programa en un archivo con otra extensión, que, dependiendo del sistema operativo puede variar. Por ejemplo, en Windows, se guardará con la extensión (.obj), abreviatura de object.

Enlace. (linkaje o montaje). Los programas pueden utilizar funciones de la biblioteca estándar de C, tales como scanf o printf. De cada una de ellas existe un código objeto que debe ser enlazado (unido) al código objeto del programa que las utilice. Esto se realiza mediante un programa llamado enlazador, montador o linkador.

Como resultado del enlace, el enlazador guardará, en disco, un archivo ejecutable. En Windows, dicho archivo tendrá extensión (.exe), abreviatura de executable. Dicho archivo será "el ejecutable".

Además de las funciones de la biblioteca estándar de C, el programador también puede utilizar funciones que hayan sido desarrolladas por él mismo. Éstas pueden agruparse en su propia biblioteca de funciones (no estándar). Por lo que, también en esta fase, el código objeto de dichas funciones deberá ser enlazado al código objeto del programa que las utilice.

Gráficamente, el proceso de puesta a punto de un programa escrito en C se puede ver en la siguiente figura:

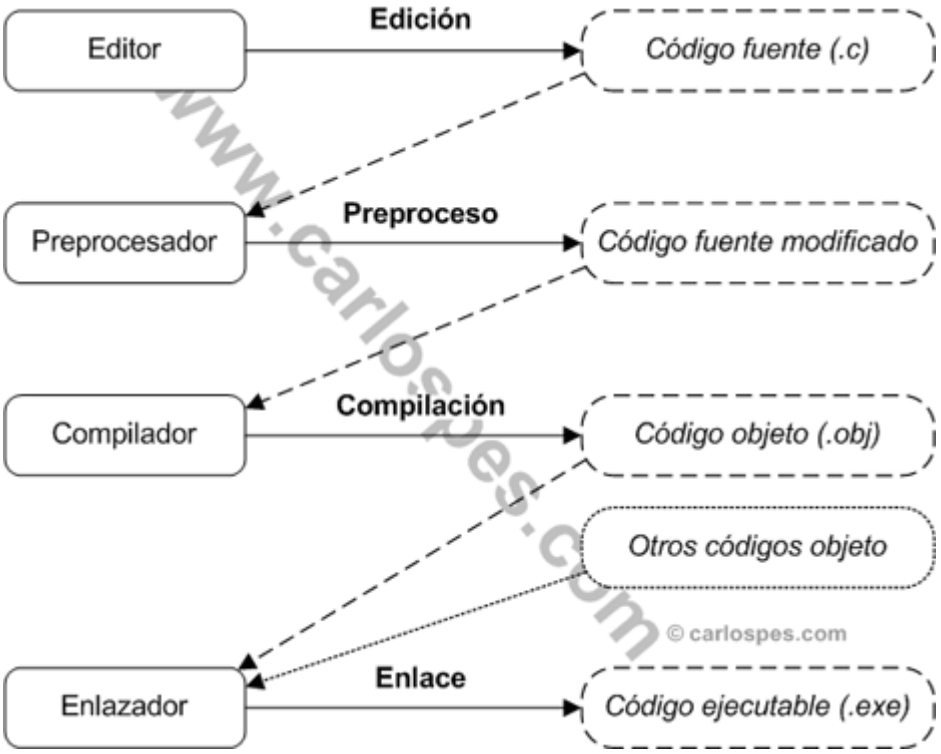


Figura - Fases de la puesta a punto de un programa en lenguaje C.

Finalmente, para que el código ejecutable de un programa se ejecute en la computadora, es necesario que un programa del sistema operativo, llamado cargador, lo lleve a la memoria principal de la misma. A partir de ese momento, la CPU empezará a ejecutarlo.

Estructura de un Programa en Lenguaje C

¿Cuál es la estructura básica de un programa en C?

Un programa escrito en lenguaje C está compuesto por una o más funciones. Como ya es sabido, una función es un programa que sirve para realizar una tarea determinada, por ejemplo, la función `scanf` sirve para leer datos desde el teclado.

Existe una función que está presente en todos los programas escritos en lenguaje C, su misión es marcar el inicio y fin de la ejecución de cada uno de ellos; es la función principal, la primera que se ejecuta; es la función `main`. Su sintaxis "básica" es:

```
int main()
{
   <bloque_de_instrucciones>
}
```

La función `main` contiene al bloque de instrucciones principal de un programa, dentro de los caracteres `abrirLlave` ({) y `cerrarLlave` (}).

Los paréntesis "`()`" escritos después de `main` sirven para indicar que el identificador `main` es una función. Es importante comprender que `main` no es una palabra reservada de C. Ningún identificador de función lo es.

Finalmente, no podemos pasar por alto que delante de `main` se ha escrito la palabra reservada `int`; enseguida veremos el porqué.

Tipos de Instrucciones en Lenguaje C

¿Qué tipos de instrucciones existen en C?

En lenguaje C, las instrucciones, también llamadas sentencias, se pueden clasificar en:

- De expresión.
- De control.
- Compuestas.

Una instrucción compuesta siempre alberga, entre llaves (`{...}`), a un <bloque_de_instrucciones>:

```
{
   <bloque_de_instrucciones>
}
```

Un bloque de instrucciones puede estar compuesto por una o más instrucciones, que, a su vez, pueden ser instrucciones de expresión, de control y/o compuestas nuevamente.

El cuerpo de toda función es una instrucción compuesta. Sirva como ejemplo el cuerpo de la función `main`, el cual alberga al bloque de instrucciones principal de un programa escrito en lenguaje C.

Las instrucciones de expresión, también conocidas como instrucciones **simples** o **elementales**, son aquellas que representan a las acciones más pequeñas (elementales) que se pueden ejecutar en un programa, y siempre van seguidas de un carácter **punto y coma** (`;`), el cual indica que la instrucción ha terminado:

```
<instrucción_de_expresión>;
```

Una instrucción de expresión es, en sí misma, una expresión. De modo que, la acción que se ejecuta es la evaluación de dicha expresión:

```
<expresión>
```

En cuanto a las instrucciones de control, existen de distintos tipos, y todas sirven para modificar el flujo de control de un programa. Veremos que, algunas de ellas deben ir seguidas del carácter **punto y coma** (`;`), pero otras no.

Mi primer Programa en Lenguaje C

¿Cómo escribir un programa en C?

Ejemplo: Se quiere escribir un programa que muestre por pantalla un mensaje de saludo:

```
Hola mundo
```

Obsérvese que, el saludo se visualiza justo en la esquina superior izquierda de la pantalla. Más adelante se estudiará cómo se puede mostrar en otro lugar de la pantalla.

En lenguaje C, el código fuente de dicho programa puede ser:

```
#include <stdio.h>

int main()
{
   printf( "Hola mundo" );

   return 0;
}
```

Todas las funciones de la biblioteca estándar de lenguaje C son subprogramas que ya están compilados, es decir, junto a cualquier compilador de lenguaje C se acompañan los códigos objeto de todas las funciones de su biblioteca estándar, pero no sus códigos fuente. Por tanto, aunque no sea posible modificar sus códigos fuente, sí se puede hacer uso de dichas funciones en cualquier programa. Por ejemplo, se puede llamar a la función `printf` para que muestre por pantalla el saludo "Hola mundo".

```
printf( "Hola mundo" );
```

Dentro de los paréntesis "`()`" de la función `printf`, se debe escribir, entre comillas dobles (`"`), el mensaje o cadena de caracteres que se desea mostrar por pantalla.

Obsérvese que, después del carácter cerrarparéntesis "`)`" se ha escrito un punto y coma (`;`), esto implica que la llamada a la función `printf` es considerada como una instrucción de expresión.

Del código fuente preprocesado de un programa, el compilador generará un código objeto que se debe unir (enlazar) con los códigos objeto de las funciones de la biblioteca estándar del lenguaje C que se llamen desde dicho programa. Por ejemplo, el código objeto de nuestro primer programa se debe enlazar con el código objeto del subprograma `printf`.

El enlazador sabe dónde encontrar el código objeto de las funciones de la biblioteca estándar de C que utilice un programa. Sin embargo, para poder utilizar una función (sea de la biblioteca estándar de C o no) en un programa, la función debe ser declarada previamente, al igual que se tienen que declarar las variables y las constantes que usa un programa.

Para que el compilador conozca la declaración de la función `printf`, hay que utilizar la directiva delpreprocesador`#include`:

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

En este caso, la directiva `#include` indica, al preprocesador, que debe incluir, antes de la compilación, en el código fuente del programa, el contenido del archivo <stdio.h> (`stdio`, Standard Input/Output). En dicho archivo están escritas las declaraciones de todas las funciones de entrada y salida estándar de la biblioteca estándar de C, como `printf`. Si el compilador no sabe quien es `printf`, no podrá generar el código objeto del programa.

Las funciones de la biblioteca estándar de C están clasificadas en base a su funcionalidad, y sus declaraciones se agrupan en archivos con extensión (.h), los cuales son llamados archivos de cabecera. Además de stdio.h, algunos de los archivos de cabecera más utilizados en lenguaje C son: math.h, string.h y stdlib.h. En ellos están escritas, respectivamente, las declaraciones de las funciones matemáticas, funciones de cadena y funciones de utilidad de la biblioteca estándar de C.

Después de `#include`, el nombre del archivo de cabecera se puede escribir entre los caracteres

menor mayor(<stdio.h>), o entre comillas dobles ("stdio.h"). Cuando se escriba entre comillas dobles, el preprocesador buscará dicho archivo en el directorio actual de trabajo, y de no encontrarlo ahí, entonces lo buscará en el directorio especificado por el compilador. En el caso de escribirse entre los caracteres menor y mayor, el proceso de búsqueda será al revés.

Además de la directiva del preprocesador `#include`, existen otras, pero, ninguna de ellas es una palabra reservada del lenguaje C.

El código objeto generado por el compilador de C tendrá "huecos" (espacios) en donde más tarde el enlazador escribirá el código objeto correspondiente a las llamadas de las funciones ya compiladas (como `printf`) y, así, generar el archivo ejecutable.

Toda función retorna un valor. En nuestro primer programa se ha escrito:

```
return 0;
```

Esto quiere decir que la función `main` devuelve el valor `0`. Precediendo a `main` se ha escrito la palabra reservada `int`, indicando así, que la función retornará un valor de tipo `int` (`entero`).

```
int main()
```

En general, la instrucción `return` suele ser la última del bloque de instrucciónes de la función `main`. Al retornar el valor `0`, indica (informa al sistema operativo) que el programa finalizó correctamente, es decir, sin producirse ningún error en su ejecución. Cuando la función `main` devuelva un valor distinto de cero, esto siginificará que se ha producido algún error en la ejecución del programa, o que ha ocurrido algo fuera de lo normal.

La instrucción `return` es una de las instrucciones de control que existen en lenguaje C. Por tanto, es una palabra reservada. Después del valor de retorno (que es una expresión) se debe escribir un punto y coma (`;`). La sintaxis de la instrucción `return` es:

```
return <expresión>;
```

Por el momento, se ha visto que la sintaxis "básica" de un programa escrito en lenguaje C es:

```
[ <directivas_del_preprocesador> ]

int main()
{
   <bloque_de_instrucciones>
}
```

Ahora es una buena ocasión para poner en práctica lo aprendido en el apartado Fases de la Puesta a punto de un Programa. Para ello, puede consultar las Guías de Uso de algunos compiladores de C/C++.

Tipos de Datos en Lenguaje C

¿Qué tipos de datos existen en C?

» Datos de Tipo Entero (int)
» Datos de Tipo Real (float o double)
» Datos de Tipo Carácter (char)
» Datos sin Valor (void)
» Datos de Tipo Lógico

» Datos de Tipo Cadena

En lenguaje C se dice que todos los datos que utilizan los programas son básicos (simples predefinidos o estándares) o derivados. Los tipos básicos en lenguaje C se clasifican en:

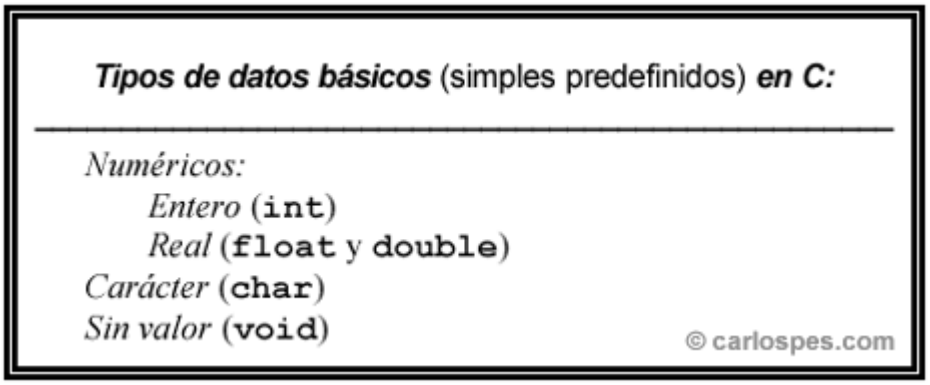


Figura - Clasificación de los tipos de datos básicos en lenguaje C.

Identificadores en Lenguaje C

¿Qué son los identificadores en C?

» Ejemplos de Identificadores válidos y no válidos
» Palabras Reservadas

La mayoría de los elementos de un programa se diferencian entre sí por su nombre. Por ejemplo, los tipos de datos básicos en lenguaje C se nombran como:

`char, int, float, double` y `void`

Cada uno de ellos es un **identificador**. Un identificador es el nombre que se le da a un elemento de un algoritmo (o programa). Por ejemplo, el tipo de dato int hace referencia a un tipo de dato que es distinto a todos los demás tipos de datos, es decir, los valores que puede tomar un dato de tipo entero, no son los mismos que los que puede tomar un dato de otro tipo.

Los identificadores `char, int, float, double` y `void` están predefinidos, forman parte del lenguaje C. No obstante, en el código de un programa también pueden existir identificadores definidos por el programador. Por ejemplo, un programa puede utilizar variables y constantes definidas por el programador.

En lenguaje C, a la hora de asignar un nombre a un elemento de un programa, se debe tener en cuenta que todo indenficador debe cumplir las siguientes reglas de sintaxis:

1. Consta de uno o más caracteres.
2. El primer carácter debe ser una letra o el carácter subrayado (_), mientras que, todos los demás pueden ser letras, dígitos o el carácter subrayado (_). Las letras pueden ser minúsculas o mayúsculas del alfabeto inglés. Así pues, no está permitido el uso de las letras 'ñ' y 'Ñ'.
3. No pueden exitir dos identificadores iguales, es decir, dos elementos de un programa no pueden nombrarse de la misma forma. Lo cual no quiere decir que un identificador no pueda aparecer más de una vez en un programa.

De la segunda regla se deduce que un identificador no puede contener caracteres especiales, salvo el carácter subrayado (_). Es importante resaltar que las vocales no pueden llevar tilde ni diéresis.

Variables en Lenguaje C

¿Cómo se declara una variable en C?

En lenguaje C hay que escribir un punto y coma (;) después de la declaración de una o más variables. Así pues, la sintaxis para declarar una variable es:

```
<tipo_de_dato> <variable> [ = <expresión> ];
```

Y para más de una variable del mismo tipo se utiliza la sintaxis:

```
<tipo_de_dato> <variable_1> [= <expresión_1>],
               <variable_2> [= <expresión_2>],
                    ...,
               <variable_n> [= <expresión_n>];
```

Una expresión representa a un valor de un tipo de dato. En el apartado Operadores y Expresiones se estudiarán en detalle las expresiones.

Constantes en Lenguaje C

¿Qué tipos de constantes existen en C?

» Constantes de Tipo Entero
» Constantes de Tipo Real
» Constantes de Tipo Carácter
» Constantes de Tipo Cadena

En lenguaje C, una constante puede ser de tipo entero, real, carácter, de cadena o enumerado. Las contantes de tipo enumerado se van a estudiar más adelante. En cuanto a las demás, se pueden expresar de dos formas diferentes:

1. Por su valor.
2. Con un nombre (identificador).

Ejemplo 1: Las siguientes contantes de tipo entero están expresadas por su valor:

```
-5
10
```

Para expresar una constante con un nombre, la constante debe ser declarada previamente. Las constantes que se declaran en un programa escrito en lenguaje C reciben un tratamiento diferente al de la mayoría de los lenguajes de programación. En C, para representar a las constantes, se utilizan constantes simbólicas. Una constante simbólica representa (sustituye) a una secuencia de caracteres, en vez de representar a un valor (dato almacenado en memoria).

Para declarar una constante simbólica, en lenguaje C, se utiliza una nueva directiva del preprocesador:

```
#define <constante> <secuencia_de_caracteres>
```

La directiva `#define` indica al preprocesador que debe sustituir, en el código fuente del programa, todas las ocurrencias del <nombre_de_la_constante> por la <secuencia_de_caracteres>, antes de la compilación.

Ejemplo 2: Dos constantes muy habituales son:

```
#define PI 3.141592 #define
NUMERO_E 2.718281
```

En programación es una buena práctica escribir los identificadores de las constantes en mayúsculas, de esta forma es más fácil localizarlos en el código de un programa. Nótese que, después de la declaración de una constante simbólica no se escribe un carácter punto y coma (`;`), cosa que sí se debe hacer al declarar una variable.

Por otra parte, no se puede declarar más de una constante simbólica en una misma línea de código.

Ejemplo 3: Para declarar las constantes simbólicas `PI` y `NUMERO_E`, no se puede escribir:

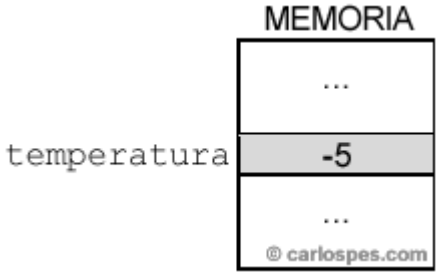
```
#define PI 3.141592, NUMERO_E 2.718281
```

Por otra parte, en C, es posible declarar una variable indicando que su valor es inalterable. Para ello, se utiliza el cualificador `const`.

Ejemplo 4: Uso de `const`:

```
const int temperatura = -5;
```

En el ejemplo, se ha declarado la variable entera temperatura inicializada al valor -5 y, por medio de const, que es una palabra reservada, se ha indicado que su valor no puede cambiar durante la ejecución del programa. En cierta manera, la variable temperatura está simulando a una constante.



Ya se sabe cómo escribir la declaración de una constante y de las variables que utilice un programa y, también, se conoce el lugar en que se tiene que escribir la directiva del preprocesador `#define` para declarar la constante.

En cuanto a las variables que utiliza un programa, su declaración puede escribirse en varios lugares, pero, de momento las vamos a declarar inmediatemente después del carácter abrir llave (`{`) de la función `main`.

```
[ <directivas_del_preprocesador> ]

int main()
{
   [ <declaraciones_de_variables> ]
```

```
   <lista_de_instrucciones>
}
```

Ejemplo 5: Si en un programa se quieren declarar dos variables (`area` y `radio`) y una constante (`PI`), se puede escribir:

```
#define PI 3.141592

int main()
{
   float area, radio;

   ...
}
```

» Ejemplos de declaración de Variables

En lenguaje C hay que escribir un punto y coma (`;`) después de la declaración de una o más variables. Así pues, la sintaxis para declarar una variable es:

```
<tipo_de_dato> <variable> [ = <expresión> ];
```

Y para más de una variable del mismo tipo se utiliza la sintaxis:

```
<tipo_de_dato> <variable_1> [= <expresión_1>],
               <variable_2> [= <expresión_2>],
                    ...,
               <variable_n> [= <expresión_n>];
```

Una expresión representa a un valor de un tipo de dato. En el apartado Operadores y Expresiones se estudiarán en detalle las expresiones.

Operadores y Expresiones en Lenguaje C

¿Qué tipos de operadores y expresiones existen en C?

» Expresiones Aritméticas
» El Operador División
» Conversión de Tipo (Casting)
» La Función pow
» Expresiones Lógicas
» Operadores Relacionales
» Operadores Lógicos
» La Función strcat
» Operadores de Asignación
» La Función strcpy
» Los Operadores Incremento (++) y Decremento (--)
» Prioridad de Operadores en Lenguaje C

En un programa, el tipo de un dato determina las operaciones que se pueden realizar con él. Por ejemplo, con los datos de tipo entero se pueden realizar operaciones aritméticas, tales como la suma, la resta o la multiplicación.

Ejemplo 1: Algunos ejemplos son:

`111 + 6` (operación suma)

`19 - 72` (operación resta)

`24 * 3` (operación multiplicación)

Todas las operaciones del ejemplo constan de dos operandos (constantes enteras) y un operador. La mayoría de las veces es así, pero, también es posible realizar operaciones con distinto número de operadores y/u operandos.

Ejemplo 2: Por ejemplo:

`111 + 6 - 8` (tres operandos y dos operadores)

`-( ( +19 ) + 72 )` (dos operandos y tres operadores)

`-( -72 )` (un operando y dos operadores)

En las operaciones del ejemplo se puede observar que los caracteres más (+) y menos (-) tienen dos usos:

1. Operadores suma y resta.
2. Signos de un número (también son operadores).

Los operadores de signo más (+) y menos (-) son operadores monarios, también llamados unarios, ya que, actúan, solamente, sobre un operando.

Los caracteres abrir paréntesis "(" y cerrar paréntesis ")" se utilizan para establecer la prioridad de los operadores, es decir, para establecer el orden en el que los operadores actúan sobre los operandos.

Un operador indica el tipo de operación a realizar sobre los operandos (datos) que actúa. Los operandos pueden ser:

- Constantes (expresadas por su valor o con un nombre (identificador)).
- Variables.
- Llamadas a funciones.
- Elementos de formaciones (arrays).

En este apartado se van a tratar operaciones en donde sólo aparecen constantes y variables. Cuando se combinan uno o más operadores con uno o más operandos se obtiene una expresión. De modo que, una expresión es una secuencia de operandos y operadores escrita bajo unas reglas de sintaxis.

Ejemplo 3: Dadas las siguientes declaraciones de constantes y variables:

```
#define PI 3.141592

int numero = 2;
float radio_circulo = 3.2;
```

Algunos ejemplos de expresiones son:

```
2 * PI * radio_circulo
```

`( PI * PI )`

`numero * 5`

De sus evaluaciones se obtienen los valores:

`20.106189` (valor real) (`2 * 3.141592 * 3.2`)

`9.869600` (valor real) (`3.141592 * 3.141592`)

`10` (valor entero) (`2 * 5`)

Un operador siempre forma parte de una expresión, en la cual, el operador siempre actúa sobre al menos un operando. Por el contrario, un operando sí puede

Prioridad de Operadores en Lenguaje C

¿Qué prioridad tienen los operadores en C?

La prioridad de todos los operadores del lenguaje C estudiados hasta el momento en este curso/tutorial es:

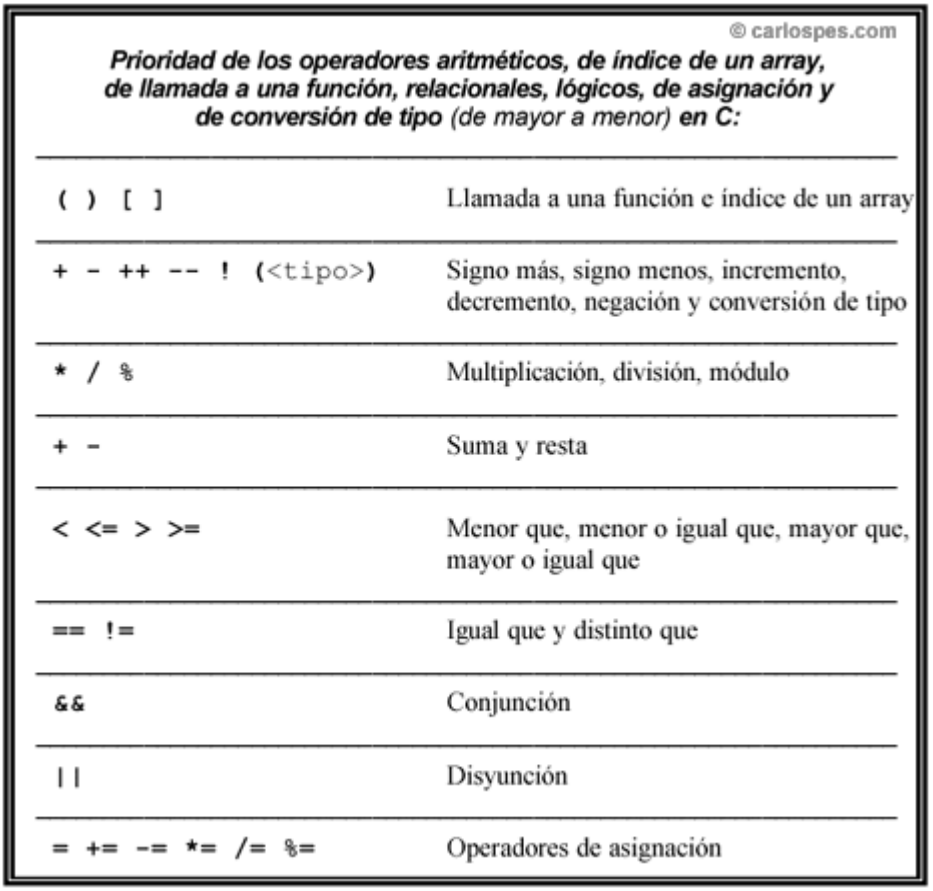


Figura - Prioridad de los operadores aritméticos, de índice de un array, de llamada a una función, relacionales, lógicos, de asignación y de conversión de tipo en C.

Obsérvese que, en lenguaje C, tanto los paréntesis "()" que se utilizan para llamar a una función, como los corchetes "[]" que albergan el índice de un array, también son considerados operadores. Además, son los operadores más prioritarios y, en una expresión, se evalúan de izquierda a derecha.

Por otra parte, los operadores incremento (++), decremento (--) y conversión de tipo "(<tipo>)" entran dentro de la categoría de operadores monarios. De manera que, al igual que los operadores signo más (+), signo menos (-), negación (!) y de asignación, todos ellos se evalúan de derecha a izquierda en una expresión, al revés que todos los demás.
parecer solo en una expresión.

En programación, de la evaluación de una expresión siempre se obtiene un valor. Dicho valor puede ser de tipo: entero, real, lógico, carácter o cadena. Por consiguiente, una expresión puede ser:

- Aritmética (devuelve un número entero o real).
- Lógica (devuelve un valor lógico: verdadero o falso)
- De carácter (devuelve un carácter representable por el ordenador).
- De cadena (devuelve una cadena).

Entrada y Salida Estándar en Lenguaje C

¿Qué son printf y scanf en C?

» La Función printf
» Texto Ordinario en la Función printf
» Especificadores de Formato en la Función printf
» Secuencias de Escape en la Función printf
» La Función scanf
» Epecificadores de Formato en la Función scanf
» Ejercicios de Entrada y Salida Estándar (scanf y printf)

En lenguaje C no existen palabras reservadas para realizar entradas y salidas. Para ello, el programador puede hacer uso de las funciones de entrada y salida estándar proporcionadas por la biblioteca estándar de lenguaje C, como son printf y scanf, entre otras que estudiaremos más adelante en este curso/tutorial.

Comentarios en Lenguaje C

¿Cómo escribir comentarios en un programa en C?

En lenguaje C, los comentarios se escriben entre los caracteres barra-asterisco (/*) y asterisco-barra(*/).

Ejemplo: Código fuente de un programa comentado:

```
/*********************************************/
/* Programa: Area_de_una_circunferencia      */
/*                                           */
/* Descripción: Recibe por teclado el radio  */
/* de una circunferencia, mostrando su área  */
/* por pantalla.                             */
/*                                           */
/* Autor: Carlos Pes                         */
/*                                           */
```

```
/* Fecha: 31/03/2005                                */
/**************************************************/

#include <math.h>
#include <stdio.h>

#define PI 3.141592 /* Definición de una constante */

int main()
{
   /* Variables del programa */
```

Instrucciones Alternativas en Lenguaje C

¿Qué tipos de instrucciones de control alternativas existen en C?

A continuación, se estudian las distintas **instrucciones de control alternativas** que se pueden utilizar en lenguaje C: doble (`if else`), simple (`if`) y múltiple (`switch`).

Instrucción de Control Alternativa Doble (if else)

» Instrucción if else en C
» Ejercicios de la Instrucción Alternativa Doble (if else) en Lenguaje C

Instrucción de Control Alternativa Simple (if)

» Instrucción if en C
» Ejercicios de la Instrucción Alternativa Simple (if) en Lenguaje C

Instrucción de Control Alternativa Múltiple (switch)

» Instrucción switch en C
» Ejercicios de la Instrucción Alternativa Múltiple (switch) en Lenguaje C

Anidamiento de Instrucciones de Control Alternativas

» Anidamiento de Instrucciones Alternativas en Lenguaje C
» Alternativa Doble (if else) en Doble (if else)
» Alternativa Múltiple (switch) en Doble (if else)
» Ejercicios de Anidamiento de Instrucciones Alternativas en Lenguaje C

```
   float area, radio;

   printf( "\n   Introduzca radio: " );
   scanf( "%f", &radio ); /* Entrada de dato */
   /* Cálculo del área de la circunferencia */
   area = PI * pow( radio, 2 );
   /* Salida por pantalla del resultado */
   printf( "\n   El area de la circunferencia es: %f", area );

   return 0;
}
```

Los comentarios serán ignorados por el compilador y, por tanto, su presencia en el código fuente es, meramente, informativa.
Instrucciones Repetitivas en Lenguaje C

¿Qué tipos de instrucciones de control repetitivas existen en C?

A continuación, se estudian las distintas instrucciones de control repetitivas que se pueden utilizar en lenguaje C: mientras (while), hacer...mientras (do while) y para (for).

Instrucción de Control Repetitiva mientras (while)

» Instrucción while en C
» Ejercicios de la Instrucción Repetitiva mientras (while) en Lenguaje C

Instrucción de Control Repetitiva hacer...mientras (do while)

» Instrucción do while en C
» Ejercicios de la Instrucción Repetitiva hacer...mientras (do..while) en Lenguaje C

Instrucción de Control Repetitiva para (for)

» Instrucción for en C
» ¿Cuándo usar un Bucle u otro?
» Ejercicios de la Instrucción Repetitiva para (for) en Lenguaje C

Anidamiento de Instrucciones de Control Repetitivas y Alternativas

» Anidamiento de Instrucciones Repetitivas y Alternativas en Lenguaje C
» Bucle for en do while
» Instrucción if en Bucle for
» Ejercicios de Anidamiento de Instrucciones de Control Repetitivas y Alternativas en Lenguaje C

Sentencia if

La construcción **`if`** sirve para ejecutar código sólo si una condición es cierta:

```
if   ( condición )
     sentencia
```

La **condición** es una expresión de cualquier clase.

- Si el resultado de la expresión es CERO,
  se considera una condición FALSA.
- Si el resultado de la expresión NO ES CERO,
  se considera una condición CIERTA.

Ejemplo:

```
int x = 1;
main()
{
```

```
  if ( x == 1 )
       printf ("la variable x vale uno\n");
  if ( x>=2 && x<=10 )
       printf ("x está entre 2 y 10\n");
}
```

Construcción else

Con la construcción **`else`** se pueden definir acciones para cuando la condición del**`if`** sea falsa.
La sintaxis es

```
if ( condición )
    sentencia
else
    sentencia
```

Ejemplo:

```
if ( x==1 )
      printf ("la variable x vale uno\n");
else
      printf ("x es distinta de uno\n");
```

Bucle while

Para ejecutar el mismo código varias veces, se puede utilizar:

```
while ( condición )
      sentencia
```

La ***sentencia*** se ejecuta una y otra vez mientras la ***condición*** sea cierta.

Ejemplos:

```
main()
{
      int x=1;
      while ( x < 100 )
      {
            printf("Línea número %d\n",x);
            x++;
      }
}
```

Ejemplo usando el operador de predecremento:

```
main()
```

```
{
       int x=10;
       while ( --x )
       {
              printf("una línea\n");
       }
}
```

En cada iteración se decrementa la variable **x** y se comprueba el valor devuelto por **`--x`**. Cuando esta expresión devuelva un cero, se abandonará el bucle. Esto ocurre después de la iteración en la que **x** vale uno.

Bucle for

También se pueden ejecutar bucles con **`for`**, según esta sintaxis:

```
       for ( expresión_inicial; condición; expresión_de_paso )
          sentencia
```

La ***expresión_inicial*** se ejecuta antes de entrar en el bucle.
Si la ***condición*** es cierta, se ejecuta ***sentencia*** y después ***expresión_de_paso***.
Luego se vuelve a evaluar la ***condición***, y así se ejecuta la sentencia una y otra vez hasta que la condición sea falsa.

El bucle **`for`** es (casi) equivalente a

```
       expresión_inicial;
       while ( condición )
       {
           sentencia
           expresión_de_paso;
       }
```

Ejemplo típico de uso:

```
int i;
...
for ( i=0; i<10; i++ )
    printf ("%d ", i );
```

Bucle do...while

Parecido al bucle **`while`**, pero iterando al menos una vez.

Sintaxis:

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

La ***sentencia*** se ejecuta al menos la primera vez; luego, mientras la ***condición*** sea cierta, se itera la sentencia.

Se recomienda no utilizar esta construcción, porque las construcciones **`while`** y**`for`** bastan para diseñar cualquier clase de bucles. Muy pocos programas hoy día usan esta construccíón.

Funciones

Las funciones son porciones de código que devuelven un valor.
Permiten descomponer el programa en módulos que se llaman entre ellos.
En C no existe diferencia entre funciones y procedimientos: a todas las subrutinas se las llama *funciones*.

La **definición** de una función especifica lo siguiente:

- nombre de la función
- número de **argumentos** que lleva y tipo de cada uno de ellos
- tipo de datos que devuelve
- Cuerpo de la función (el código que ejecuta)

Sintaxis:

tipo nombre **`(`** ***arg1*** **`,`** ***arg2*** **`,`** ***...*** **`)`**
`{`
... cuerpo ...
`}`

Cada argumento se especifica como en una declaración de variable.
El cuerpo de la función debería contener una sentencia donde se devuelve el resultado de la función, que se hace de esta forma:

`return` ***expresión*** **`;`**

La función devolverá el resultado de la ***expresión***.

Ficheros

El estándar de C contiene varias funciones para la edición de ficheros, éstas están definidas en la cabecera *stdio.h* y por lo general empiezan con la letra f, haciendo referencia a file. Adicionalmente se agrega un tipo **FILE**, el cual se usará como *apuntador a la información del fichero*. La secuencia que usaremos para realizar operaciones será la siguiente: _

```
int main(int argc, char** argv)
{
        FILE *fp;
        fp = fopen ( "fichero.in", "r" );
        if (fp==NULL) {fputs ("File error",stderr); exit (1);}
        fclose ( fp );

        return 0;
}
```

Como vemos, en el ejemplo se utilizó el *opentype* "r", que es para la lectura.

Otra cosa importante es que el lenguaje C no tiene dentro de si una estructura para el manejo de excepciones o de errores, por eso es necesario comprobar que el archivo fue abierto con éxito "if (fp == NULL)". Si **fopen** pudo abrir el archivo con éxito devuelve la referencia al archivo (FILE *), de lo contrario devuelve **NULL** y en este caso se debera revisar la direccion del archivo o los permisos del mismo. En estos ejemplos solo vamos a dar una salida con un retorno de 1 que sirve para señalar que el programa termino por un error.

feof

Esta función sirve para determinar si el cursor dentro del archivo encontró el final (**e**nd **o**f **f**ile). Existe otra forma de verificar el final del archivo que es comparar el caracter que trae **fgetc** del archivo con el macro **EOF** declarado dentro de *stdio.h*, pero este método no ofrece la misma seguridad (en especial al tratar con los archivos "binarios"). La función **feof** siempre devolverá cero (Falso) si no es encontrado **EOF** en el archivo, de lo contrario regresará un valor distinto de cero (Verdadero).

El prototipo correspondiente de feof es:

```
        int feof(FILE *fichero);
```

rewind

Literalmente significa "rebobinar", sitúa el cursor de lectura/escritura al principio del archivo.

El prototipo correspondiente de rewind es:

```
        void rewind(FILE *fichero);
```

Lectura

Un archivo generalmente debe verse como un string (una cadena de caracteres) que esta guardado en el disco duro. Para trabajar con los archivos existen diferentes formas y diferentes funciones. Las funciones que podríamos usar para leer un archivo son:

- char fgetc(FILE *archivo)
- char *fgets(char *buffer, int tamano, FILE *archivo)
- size_t fread(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);

- int fscanf(FILE *fichero, const char *formato, argumento, ...);

Las primeras dos de estas funciones son muy parecidas entre si. Pero la tercera, por el numero y el tipo de parámetros, nos podemos dar cuenta de que es muy diferente, por eso la trataremos aparte junto al **fwrite** que es su contraparte para escritura.

fgetc

Esta función lee un caracter a la vez del archivo que esta siendo señalado con el puntero ***archivo**. En caso de que la lectura sea exitosa devuelve el caracter leído y en caso de que no lo sea o de encontrar el final del archivo devuelve **EOF**.

El prototipo correspondiente de **fgetc** es:

```
        char fgetc(FILE *archivo);
```

Esta función se usa generalmente para recorrer archivos de texto. A manera de ejemplo vamos a suponer que tenemos un archivo de texto llamado "prueba.txt" en el mismo directorio en que se encuentra el fuente de nuestro programa. Un pequeño programa que lea ese archivo será:

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

int main()
{
        FILE *archivo;
        char caracter;

        archivo = fopen("prueba.txt","r");
        if (archivo == NULL)
        {
            printf("\nError de apertura del archivo. \n\n");
        }
        else
        {
            printf("\nEl contenido del archivo de prueba es \n\n");
            while((caracter = fgetc(archivo)) != EOF)
            {
                printf("%c",caracter);
            }
        }
        fclose(archivo);
        return 0;
}
```

fgets

Esta función está diseñada para leer cadenas de caracteres. Leerá hasta n-1 caracteres o hasta que lea un cambio de línea '\n' o un final de archivo EOF. En este último caso, el carácter de cambio de línea '\n' también es leído.

El prototipo correspondiente de **fgets** es:

```
        char *fgets(char *buffer, int tamaño, FILE *archivo);
```

El primer parámetro buffer lo hemos llamado así porque es un puntero a un espacio de memoria del tipo char (podríamos usar un arreglo de char). El segundo parámetro es tamaño que es el limite en cantidad de caracteres a leer para la funcion **fgets**. Y por ultimo el puntero del archivo por supuesto que es la forma en que **fgets** sabra a que archivo debe leer.

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

int main()
{
        FILE *archivo;
        char caracteres[100];

        archivo = fopen("prueba.txt","r");

        if (archivo == NULL)
                exit(1);
        else
        {
            printf("\nEl contenido del archivo de prueba es \n\n");
            while (feof(archivo) == 0)
            {
                fgets(caracteres,100,archivo);
                printf("%s",caracteres);
            }
            system("PAUSE");
        }
        fclose(archivo);
        return 0;
}
```

Este es el mismo ejemplo de antes con la diferencia de que este hace uso de **fgets** en lugar de **fgetc**. La función **fgets** se comporta de la siguiente manera, leerá del archivo apuntado por archivo los caracteres que encuentre y a ponerlos en buffer hasta que lea un caracter menos

que la cantidad de caracteres especificada en tamaño o hasta que encuentre el final de una linea (\n) o hasta que encuentre el final del archivo (**EOF**). En este ejemplo no vamos a profundizar mas que para decir que caracteres es un buffer, los pormenores seran explicados en la sección de manejo dinámico de memoria.

El beneficio de esta función es que se puede obtener una linea completa a la vez. Y resulta muy útil para algunos fines como la construcción de un parser de algún tipo de *archivo de texto*.

fread

size_t fread (void * ptr, size_t size, size_t count, FILE * stream);

Esta función lee un bloque de una "stream" de datos. Efectúa la lectura de un arreglo de elementos "count", cada uno de los cuales tiene un tamaño definido por "size". Luego los guarda en el bloque de memoria especificado por "ptr". El indicador de posición de la cadena de caracteres avanza hasta leer la totalidad de bytes. Si esto es exitoso la cantidad de bytes leídos es (size*count).

PARAMETROS:

ptr : Puntero a un bloque de memoria con un tamaño mínimo de (size*count) bytes.

size : Tamaño en bytes de cada elemento (de los que voy a leer).

count : Número de elementos, los cuales tienen un tamaño "size".

stream: Puntero a objetos FILE, que especifica la cadena de entrada.

fscanf

La función fscanf funciona igual que scanf en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

El prototipo correspondiente de **fscanf** es:

```
        int fscanf(FILE *fichero, const char *formato, argumento, ...);
```

Podemos ver un ejemplo de su uso, abrimos el documento "fichero.txt" en modo lectura y leyendo dentro de el.

```
#include <stdio.h>


int main ( int argc, char **argv )
{
        FILE *fp;
        char buffer[100];

        fp = fopen ( "fichero.txt", "r" );
        fscanf(fp, "%s" ,buffer);
        printf("%s",buffer);
```

```
        fclose ( fp );

        return 0;
}
```

Escritura

Así como podemos leer datos desde un fichero, también se pueden crear y escribir ficheros con la información que deseamos almacenar, Para trabajar con los archivos existen diferentes formas y diferentes funciones. Las funciones que podríamos usar para escribir dentro de un archivo son:

- int fputc(int caracter, FILE *archivo)
- int fputs(const char *buffer, FILE *archivo)
- size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);
- int fprintf(FILE *archivo, const char *formato, argumento, ...);

fputc

Esta función escribe un carácter a la vez del archivo que esta siendo señalado con el puntero ***archivo**. El valor de retorno es el carácter escrito, si la operación fue completada con éxito, en caso contrario será **EOF**.

El prototipo correspondiente de **fputc** es:

```
        int fputc(int carácter, FILE *archivo);
```

Mostramos un ejemplo del uso de **fputc** en un "fichero.txt", se escribira dentro del fichero hasta que presionemos la tecla **enter**.

```
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;
        char caracter;

        fp = fopen ( "fichero.txt", "a+t" ); //parametro para escritura al
final y para file tipo texto
        printf("\nIntroduce un texto al fichero: ");
        while((caracter = getchar()) != '\n')
        {
                printf("%c", fputc(caracter, fp));
        }
        fclose ( fp );
```

```
        return 0;
}
```

fputs

La función **fputs** escribe una cadena en un fichero. la ejecución de la misma no añade el carácter de retorno de línea ni el carácter nulo final. El valor de retorno es un *número no negativo* o **EOF** en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura **FILE** del fichero donde se realizará la escritura.

El prototipo correspondiente de **fputs** es:

```
        int fputs(const char *buffer, FILE *archivo)
```

para ver su funcionamiento mostramos el siguiente ejemplo:

```
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;
        char cadena[] = "Mostrando el uso de fputs en un fichero.\n";

        fp = fopen ( "fichero.txt", "r+" );
        fputs( cadena, fp );
        fclose ( fp );

        return 0;
}
```

fwrite

Esta función está pensada para trabajar con registros de longitud constante y forma pareja con **fread**. Es capaz de escribir hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada. El valor de retorno es el número de registros escritos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria de donde se obtendrán los datos a escribir, el tamaño de cada registro, el número de registros a escribir y un puntero a la estructura **FILE** del fichero al que se hará la escritura.

El prototipo correspondiente de **fwrite** es:

```
        size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE
*archivo);
```

Un ejemplo concreto del uso de **fwrite** con su contraparte **fread** y usando *funciones* es:

```c
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;

        char cadena[] = "Mostrando el uso de fwrite en un fichero.\n";
        fp = fopen ( "fichero.txt", "r+" );
        fwrite( cadena, sizeof(char), sizeof(cadena), fp ); //char
cadena[]... cada posición es de tamaño 'char'
        fclose ( fp );

        return 0;
}
```

fprintf

La función **fprintf** funciona igual que printf en cuanto a parámetros, pero la salida se dirige a un archivo en lugar de a la pantalla.

El prototipo correspondiente de **fprintf** es:

```c
        int fprintf(FILE *archivo, const char *formato, argumento, ...);
```

Podemos ver un ejemplo de su uso, abrimos el documento "fichero.txt" en modo lectura/escritura y escribimos dentro de el.

```c
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;

        char buffer[100] = "Esto es un texto dentro del fichero.";
        fp = fopen ( "fichero.txt", "r+" );
        fprintf(fp, buffer);
        fprintf(fp, "%s", "\nEsto es otro texto dentro del fichero.");

        fclose ( fp );

        return 0;
```

```
}
```

Estructuras

Una estructura contiene varios datos. La forma de definir una estructura es haciendo uso de la palabra clave **struct**. Aqui hay ejemplo de la declaracion de una estructura:

```
struct mystruct
{
    int int_member;
    double double_member;
    char string_member[25];
} variable;
```

"variable" es una instancia de "mystruct" y no es necesario ponerla aquí. Se podria omitir de la declaracion de "mystruct" y más tarde declararla usando:

```
struct mystruct variable;
```

También es una práctica muy común asignarle un alias o sinónimo al nombre de la estructura, para evitar el tener que poner "struct mystruct" cada vez. C nos permite la posibilidad de hacer esto usando la palabra clave **typedef**, lo que crea un alias a un tipo:

```
typedef struct
{
   ...
} Mystruct;
```

La estructura misma no tiene nombre (por la ausencia de nombre en la primera linea), pero tiene de alias "Mystruct". Entonces se puede usar así:

```
Mystruct variable;
```

Note que es una convencion, y una buena costumbre usar mayúscula en la primera letra de un sinónimo de tipo. De todos modos lo importante es darle algún identificador para poder hacer referencia a la estructura: podriamos tener una estructura de datos recursiva de algún tipo.

Ejemplo de una estructura :

```
/*
 *      estructura.c
 *
```

```
 *      para el wikilibro "Programación en C (fundamentos)"
 *      bajo licencia FDL, adaptado del Dominio Público
 *
 *              Nombre Miembro          Tipo
 *              Titulo                  char[30]
 *              Artista                 char[25]
 *              Precio                  float
 *              Total Canciones         int
 */

#include <stdio.h>
#include <string.h>

/* definimos una estructura para cds */
struct cd
{
     char titulo[30];
     char artista[25];
     float precio;
     int canciones;
} Cd1 = {     /* inicializamos la estructura Cd1 crea con sus valores
               * usando las definiciones iniciales*/
               "Canciones Bebe", /* titulo */
               "Pinocho", /* artista */
               12.50, /* precio */
               16 /* total canciones */
          };

int main(void)
{
        struct cd Cd2; /* definimos una nueva estructura llamado cd2 */

        /* asignamos valores a los tipos de datos del cd2 */
        strcpy(Cd2.titulo, "New Age");

        /* la forma de insertar valores a un
         * tipo char en una estructura es usando strcpy
         * de la libreria string.h
         */
        strcpy(Cd2.artista, "Old Man");
        Cd2.precio = 15.00;
```

```
        Cd2.canciones = 12;

        /* la forma de acceder a los valores de una estructura */
        /* es usando el "." despues de la definicion del dato*/
        printf("\n Cd 1");
        printf("\n Titulo: %s ", Cd1.titulo);
        printf("\n Artista: %s ", Cd1.artista);
        printf("\n Total Canciones: %d ", Cd1.canciones);
        printf("\n Precio Cd: %.2f ", Cd1.precio);

        printf("\n");
        printf("\n Cd 2");
        printf("\n Titulo: %s ", Cd2.titulo);
        printf("\n Artista: %s ", Cd2.artista);
        printf("\n Total Canciones: %d ", Cd2.canciones);
        printf("\n Precio Cd: %.2f ", Cd2.precio); /* el .2 que esta entre
%f
                                                  * sirve para mostrar
unicamente
                                                  * 2 decimales despues
del punto*/

        return 0;
}
```

Estructuras Anidadas

Una estructura puede estar dentro de otra estructura a esto se le conoce como anidamiento o estructuras anidadas. Ya que se trabajan con datos en estructuras si definimos un tipo de dato en una estructura y necesitamos definir ese dato dentro de otra estructura solamente se llama el dato de la estructura anterior.

Definamos una estructura en nuestro programa:

```
struct empleado /* creamos una estructura llamado empleado*/
{
        char nombre_empleado[25];
        char direccion[25];
        char ciudad[20];
        char provincia[20];
        long int codigo_postal;
        double salario;
```

```
}; /* las estructuras necesitan punto y coma (;) al final */
```

Y luego necesitamos una nueva estructura en nuestro programa:

```
struct cliente /* creamos una estructura llamada cliente */
{
        char nombre_cliente[25];
        char direccion[25];
        char ciudad[20];
        char provincia[20];
        long int codigo_postal;
        double saldo;
}; /* las estructuras necesitan punto y coma (;) al final */
```

Podemos ver que tenemos datos muy similares en nuestras estructuras, asi que podemos crear una sola estructura llamada **infopersona** con estos datos idénticos:

```
struct infopersona /* creamos la estructura que contiene datos parecidos
*/
{
        char direccion[25];
        char ciudad[20];
        char provincia[20];
        long int codigo_postal;
}; /* las estructuras necesitan punto y coma (;) al final */
```

Y crear las nuevas estructuras anteriores, anidando la estructura necesaria:

```
struct empleado /* se crea nuevamente la estructura */
{
        char nombre_empleado[25];
       /* creamos direcc_empleado con "struct" del tipo "estructura
infopersona" */
        struct infopersona direcc_empleado;
        double salario;
}; /* las estructuras necesitan punto y coma (;) al final */
struct cliente /* se crea nuevamente la estructura */
{
        char nombre_cliente[25];
       /* creamos direcc_cliente con "struct" del tipo "estructura
infopersona" */
```

```
        struct infopersona direcc_cliente;
        double saldo;
}; /* las estructuras necesitan punto y coma (;) al final */
```

Y acá el ejemplo completo con estructuras anidadas:

```
/*
 *      estructura2.c
 *
 *      Julio César Brizuela <brizuelaalvarado@gmail.com> 2009
 *
 *      para el wikilibro "Programación en C (fundamentos)"
 *      bajo licencia FDL, adaptado del Dominio Público
 *
 *              Nombre Miembro          Tipo
 *
 *              Titulo                          char[30]
 *              Artista                         char[25]
 *              Precio                          float
 *              Total Canciones         int
 */

#include <stdio.h>
#include <string.h>

/* creamos nuestra estructura con datos similares */
struct infopersona
{
        char direccion[25];
        char ciudad[20];
        char provincia[20];
        long int codigo_postal;
}; /* las estructuras necesitan punto y coma (;) al final */

/* creamos nuestra estructura empleado */
struct empleado
{
        char nombre_empleado[25];
        /* agregamos la estructura infopersona
         * con nombre direcc_empleado
         */
```

```
        struct infopersona direcc_empleado;
        double salario;
}; /* las estructuras necesitan punto y coma (;) al final */

/* creamos nuestra estructura cliente */
struct cliente
{
        char nombre_cliente[25];
        /* agregamos la estructura infopersona
         * con nombre direcc_cliente
         */
        struct infopersona direcc_cliente;
        double saldo;
}; /* las estructuras necesitan punto y coma (;) al final */

int main(void)
{
        /* creamos un nuevo cliente  */
        struct cliente MiCliente;

        /*inicializamos un par de datos de Micliente */
        strcpy(MiCliente.nombre_cliente,"Jose Antonio");
        strcpy(MiCliente.direcc_cliente.direccion, "Altos del Cielo");
        /* notese que se agrega direcc_cliente haciendo referencia
         * a la estructura infopersona por el dato direccion
         */

        /* imprimimos los datos */
        printf("\n Cliente: ");
        printf("\n Nombre: %s", MiCliente.nombre_cliente);
        /* notese la forma de hacer referencia al dato */
        printf("\n Direccion: %s", MiCliente.direcc_cliente.direccion);

        /* creamos un nuevo empleado  */
        struct empleado MiEmpleado;

        /*inicializamos un par de datos de MiEmplado */
        strcpy(MiEmpleado.nombre_empleado,"Miguel Angel");
        strcpy(MiEmpleado.direcc_empleado.ciudad,"Madrid");
        /* para hacer referencia a ciudad de la estructura infopersona
         * utilizamos direcc_empleado que es una estructura anidada
```

```
         */

        /* imprimimos los datos */
        printf("\n");
        printf("\n Empleado: ");
        printf("\n Nombre: %s", MiEmpleado.nombre_empleado);
        /* notese la forma de hacer referencia al dato */
        printf("\n Ciudad: %s", MiEmpleado.direcc_empleado.ciudad);

        return 0;
}
```

Uniones

La definición de "unión" es similar a la de "estructura", La diferencia entre las dos es que en una estructura, los miembros ocupan diferentes áreas de la memoria, pero en una unión, los miembros ocupan la misma área de memoria. Entonces como ejemplo:

```
  union {
      int i;
      double d;
  } u;
```

El programador puede acceder a través de "u.i" o de "u.d", pero no de ambos al mismo tiempo. Como "u.i" y "u.d" ocupan la misma área de memoria, modificar uno modifica el valor del otro, algunas veces de maneras impredecibles.

El tamaño de una unión es el de su miembro de mayor tamaño.

Ejemplo de una unión:

```
/*
 *      uniones.c
 *
 *      Julio César Brizuela <brizuelaalvarado@gmail.com> 2009
 *
 *      para el wikilibro "Programación en C (fundamentos)"
 *      bajo licencia FDL, adaptado del Dominio Público
 */

#include <stdio.h>
#include <string.h>

/*Creamos una union*/
```

```c
union frases
{
        char mensajes[50];
        char ayudas[50];
        char lineas[50];
} palabra;

/*Creamos una estructura*/
struct comparte
{
        char mensajes[50];
        char ayudas[50];
        char lineas[50];
}Sistema;

/*Nótese que la estructura y la union tienen los mismos tipos de datos*/

int main(int argc, char** argv)
{
        /*Inicializamos*/
        strcpy(palabra.mensajes, "Primer Mensaje");
        /*Inicializamos*/
        strcpy(palabra.ayudas, "Una Ayuda");
        printf("\nFrases en Union: ");
        /*Imprimimos mensajes de union*/
        printf("\n1- %s", palabra.mensajes);
        /*Imprimimos ayudas de union*/
        printf("\n2- %s", palabra.ayudas);
        /*Inicializamos*/
        strcpy(Sistema.mensajes, "Primer Mensaje");
        /*Inicializamos*/
        strcpy(Sistema.ayudas, "Una Ayuda");

        /* Podemos notar que aunque inicializamos los valores
         * al imprimir se tiene el mismo valor para cada miembro
         * de la estructura, esto se debe a que las uniones usan el
         * mismo espacio de memoria para todos los elementos
         * de la union, siendo del tamaño de su miembro de
         * mayor tamaño, en este caso 50 bytes.
         * Entonces los tres miembros creados dentro de la
         * union comparten esos 50 bytes.
```

```
         * Entonces el ultimo valor agregado a la union es
         * el que se tiene.
         */

        printf("\n\nFrases en Struct: ");
        /*Imprimimos mensajes de struct*/
        printf("\n1- %s", Sistema.mensajes);
        /*Imprimimos ayudas de union*/
        printf("\n2- %s", Sistema.ayudas);

        /* En la estructura comparte, se reservan 50 bytes
         * de memoria para los tres miembros, en este caso
         * cada uno es independiente en memoria, asi pues se
         * puede inicializar cada uno o usar como un campo
         * independiente.
         */

        return 0;
}
```

Enumeraciones

Una enumeracion (enum) es un tipo definido con constante de tipo entero. En la declaracion de un tipo enum creamos una lista de tipo de datos que se asocian con las constantes enteras 0, 1, 2, 3, 4, 5...

su forma de definirlas es la siguiente:

```
enum
{
        enumerador1, enumerador2, … enumeradorn
};

enum Nombre
{
        enumerador1, enumerador2, … enumeradorn
};
```

En este caso al ser declaradas enumerador1 toma el valor entero de 0, enumerador2 el valor de 1 y asi sucesivamente para cada una de las expresiones siguientes.

Al declarar la enum se puede asociar a los tipos de datos a valores constantes en vez de la asociacion que por defecto se realiza (0, 1, 2, …), se utiliza entonces este formato:

```
enum Nombre
{
        enumerador1 = valor_constante1,
        enumerador2 = valor_constante2,
        ...
        enumeradorn = valor_constanten,
};
```

Un ejemplo de una enum:

```
enum Boolean
{
        FALSE,
        TRUE
};
```

Se definen dos constantes para las constantes true y false con valores iguales a 0 para False y 1 para True.

Ejemplo:

```
/*
 *      Enum.c
 *
 *      Julio César Brizuela <brizuelaalvarado@gmail.com> 2009
 *
 *      para el wikilibro "Programación en C (fundamentos)"
 *      bajo licencia FDL, adaptado del Dominio Público
 */

#include <stdio.h>

enum Boolean
{
        FALSE, TRUE
};
/* Se define un enum para emular las constantes
 * True y False con valores de 0 y 1.
 * Notese que las enum no necesitan ; al final
 * de cada tipo de dato.
 */
```

```c
/* Definimos una funcion del tipo enum llamada numero*/
enum Boolean numero(char c);

int main(int argc, char** argv)
{
	char caracter;
	int Numeros = 0;

	printf("\nIntroduce un texto. Para terminar: Enter. \n\t");
	/* Tenemos un while que mientras no se presione Enter
	 * seguira leyendo un tipo de dato caracter
	 */
	while((caracter = getchar()) != '\n')
	{
		if (numero(caracter))
		{
			Numeros++;
		}
	}
	printf("\nTotal de Numeros leidos: %d", Numeros);

	return 0;
}

enum Boolean numero(char c)
{
	switch(c)
	{
		case '0':
		case '1':
		case '2':
		case '3':
		case '4':
		case '5':
		case '6':
		case '7':
		case '8':
		case '9':
			return TRUE;
```

```
                /* Mientras el caracter valga de 0 a 9 retornara TRUE (1)
*/
                default:
                        return FALSE;
                /* Por default retornara FALSE (0) */
        }
}
```

En la siguiente enum se declaran las variables inicializando la primera y las demas con los siguientes valores enteros:

```
/*
 *      Enum2.c
 *
 *      Julio César Brizuela <brizuelaalvarado@gmail.com> 2009
 *
 *      para el wikilibro "Programación en C (fundamentos)"
 *      bajo licencia FDL, adaptado del Dominio Público
 */

#include <stdio.h>

enum DiasSemanas
{
        Domingo = 1,
        Lunes,
        Marte,
        Miercoles,
        Jueves,
        Viernes,
        Sabado
};
/* Podemos inicializar nuestra primer constante Domingo
 * en 2, asi pues las demas los siguientes valores enteros.
 */

int main(int argc, char** argv)
{
        enum DiasSemanas dia;

        for (dia = Domingo; dia <= Sabado; dia++)
```

```
        {
                printf("%d ", dia); /* Salida: 1 2 3 4 5 6 7  */
        }

        return 0;
}
```

A los enumeradores se pueden asignar valores o expresiones constantes durante la declaracion:

```
enum Hexaedro
{
        VERTICE = 8,
        LADOS = 12,
        CARAS = 6
};
```

Bibliografía

- Kernighan B. & Ritchie D. (1995). Lenguaje de Programación C (2ª. ed). México: Pearson Editorial

- Joyanes A. & Zahonero I. (2005). Programación en C (2ª ed). Mc Graw Hill

- Deitel, H. & Deitel, P. (2004) C, How to program, (7th ed). USA: Pearson Education