

# Manual de C

Héctor Tejeda Villela

---

- [Índice General](#)
- [1. Compilación de un programa en C/C++](#)
  - [1.1 Creación, compilación y ejecución de un programa](#)
    - [1.1.1 Creación del programa](#)
    - [1.1.2 Compilación](#)
    - [1.1.3 Ejecución del programa](#)
  - [1.2 El modelo de compilación de C](#)
  - [1.3 El preprocesador](#)
  - [1.4 Compilador de C](#)
  - [1.5 Ensamblador](#)
  - [1.6 Ligador](#)
  - [1.7 Algunas opciones útiles del compilador](#)
  - [1.8 Uso de las bibliotecas](#)
  - [1.9 Ejemplos](#)
    - [1.9.1 Creación de una biblioteca estática](#)
    - [1.9.2 Creación de una biblioteca compartida](#)
  - [1.10 Funciones de la biblioteca de UNIX](#)
    - [1.10.1 Encontrando información acerca de las bibliotecas.](#)
  - [1.11 Ejercicios](#)
- [2. Principios de C](#)
  - [2.1 Orígenes del C](#)
  - [2.2 Características de C](#)
  - [2.3 Estructura de un programa en C](#)
  - [2.4 Variables](#)
    - [2.4.1 Definición de variables globales](#)
    - [2.4.2 Lectura y escritura de variables](#)
  - [2.5 Constantes](#)
  - [2.6 Operadores Aritméticos](#)
  - [2.7 Operadores de Comparación](#)
  - [2.8 Operadores lógicos](#)
  - [2.9 Orden de precedencia](#)
  - [2.10 Ejercicios](#)
- [3. Estructuras Condicionales](#)
  - [3.1 La sentencia `if`](#)
  - [3.2 El operador `?`](#)
  - [3.3 La sentencia `switch`](#)
  - [3.4 Ejercicios](#)

- 4. Iteración
  - 4.1 La sentencia for
  - 4.2 La sentencia while
  - 4.3 La sentencia do-while
  - 4.4 Uso de break y continue
  - 4.5 Ejercicios
  
- 5. Arreglos y cadenas
  - 5.1 Arreglos unidimensionales y multidimensionales
  - 5.2 Cadenas
  - 5.3 Ejercicios
  
- 6. Funciones
  - 6.1 Funciones void
  - 6.2 Funciones y arreglos
  - 6.3 Prototipos de funciones
  - 6.4 Ejercicios
  
- 7. Más tipos de datos
  - 7.1 Estructuras
    - 7.1.1 Definición de nuevos tipos de datos
  - 7.2 Uniones
  - 7.3 Conversión de tipos (casts)
  - 7.4 Enumeraciones
  - 7.5 Variables estáticas
  - 7.6 Ejercicios
  
- 8. Apuntadores
  - 8.1 Definición de un apuntador
  - 8.2 Apuntadores y Funciones
  - 8.3 Apuntadores y arreglos
  - 8.4 Arreglos de apuntadores
  - 8.5 Arreglos multidimensionales y apuntadores
  - 8.6 Inicialización estática de arreglos de apuntadores
  - 8.7 Apuntadores y estructuras
  - 8.8 Fallas comunes con apuntadores
  - 8.9 Ejercicios
  
- 9. Asignación dinámica de memoria y Estructuras dinámicas
  - 9.1 Uso de malloc, sizeof y free
  - 9.2 calloc y realloc
  - 9.3 Listas ligadas
  - 9.4 Programa de revisión
  - 9.5 Ejercicios
  
- 10. Tópicos avanzados con apuntadores
  - 10.1 Apuntadores a apuntadores
  - 10.2 Entrada en la línea de comandos

- [10.3 Apuntadores a funciones](#)
- [10.4 Ejercicios](#)
  
- [11. Operadores de bajo nivel y campos de bit](#)
  - [11.1 Operadores sobre bits](#)
  - [11.2 Campos de bit](#)
    - [11.2.1 Portabilidad](#)
  - [11.3 Ejercicios](#)
  
- [12. El preprocesador de C](#)
  - [12.1 Directivas del preprocesador](#)
    - [12.1.1 #define](#)
    - [12.1.2 #undef](#)
    - [12.1.3 #include](#)
    - [12.1.4 #if Inclusión condicional](#)
  - [12.2 Control del preprocesador del compilador](#)
  - [12.3 Otras directivas del preprocesador](#)
  - [12.4 Ejercicios](#)
  
- [13. C, UNIX y las bibliotecas estándar](#)
  - [13.1 Ventajas del usar UNIX con C](#)
  - [13.2 Uso de funciones de bibliotecas y llamadas del sistema](#)
  
- [14. Biblioteca <stdlib.h>](#)
  - [14.1 Funciones aritméticas](#)
  - [14.2 Números aleatorios](#)
  - [14.3 Conversión de cadenas](#)
  - [14.4 Búsqueda y ordenamiento](#)
  - [14.5 Ejercicios](#)
  
- [15. Biblioteca <math.h>](#)
  - [15.1 Funciones matemáticas](#)
  - [15.2 Constantes matemáticas](#)
  
- [16. Entrada y salida \(E/S\) stdio.h](#)
  - [16.1 Reportando errores](#)
    - [16.1.1 perror\(\)](#)
    - [16.1.2 errno](#)
    - [16.1.3 exit](#)
  - [16.2 Flujos](#)
    - [16.2.1 Flujos predefinidos](#)
  - [16.3 E/S Basica](#)
  - [16.4 E/S formateada](#)
    - [16.4.1 printf](#)
    - [16.4.2 scanf](#)
  - [16.5 Archivos](#)
    - [16.5.1 Lectura y escritura de archivos](#)
  - [16.6 sprintf y sscanf](#)
    - [16.6.1 Petición del estado del flujo](#)

- [16.7 E/S de bajo nivel o sin almacenamiento intermedio](#)
- [16.8 Ejercicios](#)
- [17. Manejo de cadenas <string.h>](#)
  - [17.1 Funciones básicas para el manejo de cadenas](#)
    - [17.1.1 Búsqueda en cadenas](#)
  - [17.2 Prueba y conversión de caracteres <ctype.h>](#)
  - [17.3 Operaciones con la memoria <memory.h>](#)
  - [17.4 Ejercicios](#)
- [18. Acceso de Archivos y llamadas al sistema de directorios](#)
  - [18.1 Funciones para el manejo de directorios <unistd.h>](#)
    - [18.1.1 Búsqueda y ordenamiento de directorios: `sys/types.h, sys/dir.h`](#)
  - [18.2 Rutinas de manipulación de archivos: `unistd.h, sys/types.h, sys/stat.h`](#)
    - [18.2.1 Permisos de accesos a archivos](#)
    - [18.2.2 Estado de un archivo](#)
    - [18.2.3 Manipulación de archivos: `stdio.h, unistd.h`](#)
    - [18.2.4 Creación de archivos temporales: `<stdio.h>`](#)
  - [18.3 Ejercicios](#)
- [19. Funciones para el tiempo](#)
  - [19.1 Funciones básicas para el tiempo](#)
  - [19.2 Ejemplos de aplicaciones de funciones del tiempo.](#)
    - [19.2.1 Ejemplo 1: Tiempo \(en segundos\) para hacer algún cálculo.](#)
    - [19.2.2 Ejemplo 2: Inicializar la semilla de un número aleatorio.](#)
  - [19.3 Ejercicios](#)
- [20. Control de procesos: <stdlib.h>, <unistd.h>](#)
  - [20.1 Ejecutando comandos de UNIX desde C](#)
  - [20.2 `execl\(\)`](#)
  - [20.3 `fork\(\)`](#)
  - [20.4 `wait\(\)`](#)
  - [20.5 `exit\(\)`](#)
  - [20.6 Ejercicios](#)
- [21. Compilación de Programas con Archivos Múltiples](#)
  - [21.1 Archivos Cabezera](#)
  - [21.2 Variables y Funciones Externas](#)
    - [21.2.1 Alcance de las variables externas](#)
  - [21.3 Ventajas de Usar Varios Archivos](#)
  - [21.4 Como dividir un programa en varios archivos](#)
  - [21.5 Organización de los Datos en cada Archivo](#)
  - [21.6 La utilería Make](#)
    - [21.6.1 Programando Make](#)
  - [21.7 Creación de un Archivo Make \(Makefile\)](#)
  - [21.8 Uso de macros con Make](#)
  - [21.9 Ejecución de Make](#)

- [22. Comunicación entre procesos \(IPC Interprocess Communication\), PIPES](#)
  - [22.1 Entubando en un programa de C <stdio.h>](#)
    - [22.1.1 popen \(\) Tubería formateada](#)
    - [22.1.2 pipe \(\) Tubería de bajo nivel](#)
- [23. Sockets](#)
  - [23.1 Creación y nombrado de sockets](#)
  - [23.2 Conectando sockets de flujo](#)
    - [23.2.1 Transferencia de datos en un flujo y cerrado](#)
- [Sobre este documento...](#)



---

Última modificación : 2005-08-12

[Héctor Tejeda V](#)

 [htejeda@fismat.umich.mx](mailto:htejeda@fismat.umich.mx)

## Subsecciones

- [1.1 Creación, compilación y ejecución de un programa](#)
  - [1.1.1 Creación del programa](#)
  - [1.1.2 Compilación](#)
  - [1.1.3 Ejecución del programa](#)
- [1.2 El modelo de compilación de C](#)
- [1.3 El preprocesador](#)
- [1.4 Compilador de C](#)
- [1.5 Ensamblador](#)
- [1.6 Ligador](#)
- [1.7 Algunas opciones útiles del compilador](#)
- [1.8 Uso de las bibliotecas](#)
- [1.9 Ejemplos](#)
  - [1.9.1 Creación de una biblioteca estática](#)
  - [1.9.2 Creación de una biblioteca compartida](#)
- [1.10 Funciones de la biblioteca de UNIX](#)
  - [1.10.1 Encontrando información acerca de las bibliotecas.](#)
- [1.11 Ejercicios](#)



---

# 1. Compilación de un programa en C/C++

En esta capítulo se dan los procesos básicos que se requieren para compilar un programa de C. Se describe también el modelo de compilación de C y también como C soporta bibliotecas adicionales.

## 1.1 Creación, compilación y ejecución de un programa

### 1.1.1 Creación del programa

Se puede crear un archivo que contenga el programa completo, como en los ejemplos que se tienen más adelante. Se puede usar cualquier editor de textos ordinario con el que se este familiarizado. Un editor disponible en la mayoría de los sistemas UNIX es `vi`, y en Linux se puede usar `pico`.

Por convención el nombre del archivo debe terminar con ``.c'` por ejemplo: miprograma.c progprueba.c. El contenido del archivo deberá obedecer la sintaxis de C.`

## 1.1.2 Compilación

Existen muchos compiladores de C. El `cc` es el compilador estándar de Sun. El compilador GNU de C es `gcc`, el cual es bastante popular y esta disponible en varias plataformas.

Existen también compiladores equivalentes de C++ los cuales usualmente son nombrados como `CC`. Por ejemplo, Sun provee `CC` y GNU `GCC`. El compilador de GNU es también denotado como `g++`.

Existen otros compiladores menos comunes de C y C++. En general todos los compiladores mencionados operan esencialmente de la misma forma y comparten muchas opciones comunes en la línea de opciones. Más adelante se listan y se dan ejemplos de opciones comunes de los compiladores. Sin embargo, la mejor referencia de cada compilador es a través de las páginas en línea, del manual del sistema. Por ejemplo: `man gcc`.

Para compilar el programa usaremos el comando `gcc`. El comando deberá ser seguido por el nombre del programa en C que se quiere compilar. Un determinado número de opciones del compilador pueden ser indicadas también. Por el momento no haremos uso de estas opciones todavía, se irán comentando algunas más esenciales.

Por lo tanto, el comando básico de compilación es:

```
gcc programa.c
```

donde *programa.c* es el nombre del archivo.

Si hay errores obvios en el programa (tales como palabras mal escritas, caracteres no tecleados u omisiones de punto y coma), el compilador se detendrá y los reportará.

Podría haber desde luego errores lógicos que el compilador no podrá detectar. En el caso que esta fuera la situación se le estará indicando a la computadora que haga las operaciones incorrectas.

Cuando el compilador ha terminado con éxito, la versión compilada, o el ejecutable, es dejado en un archivo llamado *a.out*, o si la opción `-o` es usada con el compilador, el nombre después de `-o` es el nombre del programa compilado.

Se recomienda y es más conveniente usar la opción `-o` con el nombre del archivo ejecutable como se muestra a continuación:

```
gcc -o programa programa.c
```

el cual pone el programa compilado en el archivo del programa señalado, en éste caso en *programa*, en vez del archivo *a.out*.

## 1.1.3 Ejecución del programa

El siguiente estado es correr el programa ejecutable. Para correr un ejecutable en UNIX, simplemente se escribe el nombre del archivo que lo contiene, en este caso *programa* (o *a.out*).

Con lo anterior, se ejecuta el programa, mostrando algún resultado en la pantalla. En éste estado,

podría haber errores en tiempo de ejecución (*run-time errors*), tales como división por cero, o bien, podrían hacerse evidentes al ver que el programa no produce la salida correcta.

Si lo anterior sucede, entonces se debe regresar a editar el archivo del programa, recompilarlo, y ejecutarlo nuevamente.

## 1.2 El modelo de compilación de C

En la figura [1.1](#) se muestran las distintas etapas que cubre el compilador para obtener el código ejecutable.

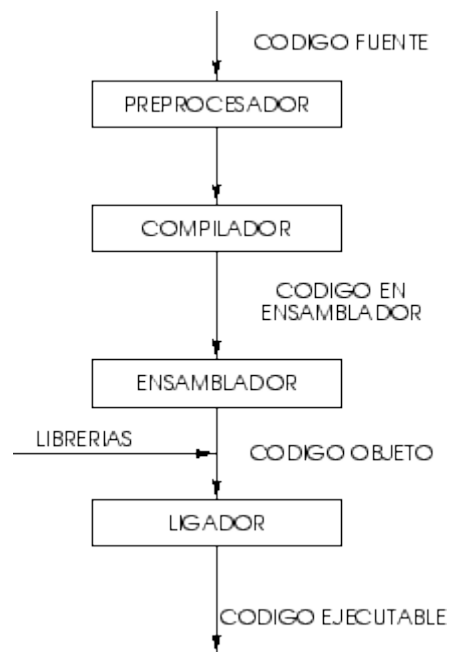


Figura 1.1: Modelo de compilación de C.

## 1.3 El preprocesador

Esta parte del proceso de compilación será cubierta con más detalle en el capítulo [12](#) referente al preprocesador. Sin embargo, se da alguna información básica para algunos programas de C.

El preprocesador acepta el código fuente como entrada y es responsable de:

- quitar los comentarios
- interpretar las *directivas del preprocesador* las cuales inician con #.

Por ejemplo:

- `#include` -- incluye el contenido del archivo nombrado. Estos son usualmente llamados archivos de *cabecera (header)*. Por ejemplo:
  - `#include <math.h>` -- Archivo de la biblioteca estándar de matemáticas.
  - `#include <stdio.h>` -- Archivo de la biblioteca estándar de Entrada/Salida.

- `#define` -- define un nombre simbólico o constante. Sustitución de macros.
  - `#define TAM_MAX_ARREGLO 100`

## 1.4 Compilador de C

El compilador de C traduce el código fuente en código de ensamblador. El código fuente es recibido del preprocesador.

## 1.5 Ensamblador

El ensamblador crea el código fuente o los archivos objeto. En los sistemas con UNIX se podrán ver los archivos con el sufijo `.o`.

## 1.6 Ligador

Si algún archivo fuente hace referencia a funciones de una biblioteca o de funciones que están definidas en otros archivos fuentes, el *ligador* combina estas funciones (con `main()`) para crear un archivo ejecutable. Las referencias a variables externas en esta etapa son resueltas.

## 1.7 Algunas opciones útiles del compilador

Descrito el modelo básico de compilación, se darán algunas opciones útiles y algunas veces esenciales. De nueva cuenta, se recomienda revisar las páginas de `man` para mayor información y opciones adicionales.

`-E`

Se compilador se detiene en la etapa de preprocesamiento y el resultado se muestra en la salida estándar.

```
gcc -E arch1.c
```

`-c`

Suprime el proceso de ligado y produce un archivo `.o` para cada archivo fuente listado. Después los archivos objeto pueden ser ligados por el comando `gcc`, por ejemplo:

```
gcc arch1.o arch2.o ... -o ejecutable
```

`-lbiblioteca`

Liga con las bibliotecas objeto. Esta opción deberá seguir los argumentos de los

archivos fuente. Las bibliotecas objeto son guardadas y pueden estar estandarizadas, un tercero o usuario las crea. Probablemente la biblioteca más comúnmente usada es la biblioteca matemática (`math.h`). Esta biblioteca deberá ligarse explícitamente si se desea usar las funciones matemáticas (y por supuesto no olvidar el archivo cabecera `#include <math.h>`, en el programa que llama a las funciones), por ejemplo:

```
gcc calc.c -o calc -lm
```

Muchas otras bibliotecas son ligadas de esta forma.

### **-Ldirectorio**

Agrega directorios a la lista de directorios que contienen las rutinas de la biblioteca de objetos. El ligador siempre busca las bibliotecas estándares y del sistema en `/lib` y `/usr/lib`. Si se quieren ligar bibliotecas personales o instaladas por usted, se tendrá que especificar donde estan guardados los archivos, por ejemplo:

```
gcc prog.c -L/home/minombr/mislibs milib.a
```

### **-Itrayectoria**

Agrega una trayectoria o ruta a la lista de directorios en los cuales se buscarán los archivos cabecera `#include` con nombres relativos (es decir, los que no empiezan con diagonal `/`).

El procesador por default, primero busca los archivos `#include` en el directorio que contiene el archivo fuente, y después en los directorios nombrados con la opción `-I` si hubiera, y finalmente, en `/usr/include`. Por lo tanto, si se quiere incluir archivos de cabecera guardados en `/home/minombr/miscabeceras` se tendrá que hacer:

```
gcc prog.c -I/home/minombr/miscabeceras
```

**Nota:** Las cabeceras de las bibliotecas del sistema son guardados en un lugar especial (`/usr/include`) y no son afectadas por la opción `-I`. Los archivos cabecera del sistema y del usuario son incluidos en una manera un poco diferente.

### **-g**

Opción para llamar las opciones de depuración (debug). Instruye al compilador para producir información adicional en la tabla de símbolos que es usado por una variedad de utilerías de depuración. Por ejemplo, si se emplea el depurador de GNU, el programa deberá compilarse de la siguiente forma para generar extensiones de GDB:

```
gcc -ggdb -o prog prog.c
```

### **-D**

Define símbolos como identificadores (`-Didentificador`) o como valores (`-Dsímbolo=valor`) en una forma similar a la directiva del preprocesador `#define`).

Muestra en la salida estandar de errores los comandos ejecutados en las etapas de compilación.

## 1.8 Uso de las bibliotecas

C es un lenguaje extremadamente pequeño. Muchas de las funciones que tienen otros lenguajes no están en C, por ejemplo, no hay funciones para E/S, manejo de cadenas o funciones matemáticas.

La funcionalidad de C se obtiene a través de un rico conjunto de bibliotecas de funciones.

Como resultado, muchas implementaciones de C incluyen bibliotecas *estándar* de funciones para varias finalidades. Para muchos propósitos básicos estas podrían ser consideradas como parte de C. Pero pueden variar de máquina a máquina.

Un programador puede también desarrollar sus propias funciones de biblioteca e incluso bibliotecas especiales de terceros, por ejemplo, NAG o PHIGS.

Todas las bibliotecas (excepto E/S estándar) requieren ser explícitamente ligadas con la opción `-l` y, posiblemente con `L`, como se señalo previamente.

## 1.9 Ejemplos

### 1.9.1 Creación de una biblioteca estática

Si se tiene un conjunto de rutinas que se usen en forma frecuente, se podría desear agruparlas en un conjunto de archivos fuente, compilar cada archivo fuente en un archivo objeto, y entonces crear una biblioteca con los archivos objeto. Con lo anterior se puede ahorrar tiempo al compilar en aquellos programas donde sean usadas.

Supongamos que se tiene un conjunto de archivos que contengan rutinas que son usadas frecuentemente, por ejemplo un archivo `cubo.c`:

```
float cubo(float x)
{
    return (x*x*x);
}
```

y otro archivo `factorial.c`

```
int factorial(int n)
{
    int i, res=1;
    for(i=1; i<=n; i++)
        res*=i;
    return (res);
}
```

```
}
```

Para los archivos de nuestras funciones también se debe tener un archivo de cabecera, para que puedan ser usadas, suponiendo que se tiene el siguiente archivo `libmm.h` con el siguiente contenido:

```
extern float cubo(float);  
extern int factorial(int);
```

El código que use la biblioteca que se esta creando podría ser:

```
/* Programa prueba.c */  
#include "libmm.h"  
#define VALOR 4  
  
main()  
{  
    printf("El cubo de %d es %f\n",VALOR, cubo(VALOR) );  
    printf("\t y su factorial es %d\n",factorial(VALOR) );  
}
```

Para crear la biblioteca se deben compilar los archivos fuente, que lo podemos hacer de la siguiente forma:

```
$ gcc -c cubo.c factorial.c
```

Lo cual nos dejará los archivos `cubo.o` y `factorial.o`. Después se debe crear la biblioteca con los archivos fuentes, suponiendo que nuestra biblioteca se llame `libmm.a`, tendrás que hacerlo con el comando `ar` así:

```
$ ar r libmm.a cubo.o factorial.o
```

Cuando se actualiza una biblioteca, se necesita borrar el archivo anterior (`libmm.a`). El último paso es crear un índice para la biblioteca, lo que permite que el ligador pueda encontrar las rutinas. Lo anterior, lo hacemos con el comando `ranlib`, por lo que teclearemos ahora:

```
$ ranlib libmm.a
```

Los últimos dos pasos pudieron ser combinados en uno sólo, entonces hubieramos podido teclear:

```
$ar rs libmm.a cubo.o factorial.o
```

Ahora que ya tenemos la biblioteca, es conveniente que coloquemos nuestra biblioteca y el archivo cabecera en algún lugar apropiado. Supongamos que dejamos la biblioteca en `~/lib` y el fichero cabecera en `~/include`, debemos hacer lo siguiente:

```
$ mkdir ../include  
$ mkdir ../lib  
$ mv libmm.h ../include  
$ mv libmm.a ../lib
```

Si llegarás a modificar la biblioteca, tendrías que repetir la última instrucción.

Se debe ahora compilar el archivo con la biblioteca, de la siguiente forma:

```
gcc -I../include -L../lib -o prueba prueba.c -lmm
```

## 1.9.2 Creación de una biblioteca compartida

Las ventajas que presentan las bibliotecas compartidas, es la reducción en el consumo de memoria, si son usadas por más de un proceso, además de la reducción del tamaño del código ejecutable. También se hace el desarrollo más fácil, ya que cuando se hace algún cambio en la biblioteca, no se necesita recompilar y reenlazar la aplicación cada vez. Se requiere lo anterior sólo si se modifico el número de argumentos con los que se llama una función o se cambio el tamaño de alguna estructura.

El código de la biblioteca compartida necesita ser independiente de la posición, para hacer posible que sea usado el código por varios programas. Para crear la biblioteca hacerlo de la siguiente forma:

```
$ gcc -c -fPIC cubo.c factorial.c
```

Para generar la biblioteca dinámica hacer lo siguiente:

```
$ gcc -shared -o libmm.so cubo.o factorial.o
```

No existe un paso para la indexación como ocurre en las bibliotecas estáticas.

Después habrá que mover la biblioteca dinámica a su directorio correspondiente (`../lib`) y proceder a compilar para que nuestro código use la biblioteca.

```
$ gcc -I../include -L../lib -o prueba prueba.c -lmm
```

Nos preguntamos que sucede si hay una biblioteca compartida (`libmm.so`) y una estática (`libmm.a`) disponibles. En este caso, el ligador siempre toma la compartida. Si se desea hacer uso de la estática, se tendrá que nombrar explícitamente en la línea de comandos:

```
$ gcc -I../include -L../lib -o prueba prueba.c libmm.a
```

Cuando se usan bibliotecas compartidas un comando útil es `ldd`, el cual nos informa que bibliotecas compartidas un programa ejecutable usa, a continuación un ejemplo:

```
$ ldd prueba
    libstuff.so => libstuff.so (0x40018000)
    libc.so.6 => /lib/i686/libc.so.6 (0x4002f000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Como se ve en cada línea aparece el nombre de la biblioteca, el camino completo a la biblioteca que es usada, y donde en el espacio de direcciones virtuales la biblioteca esta mapeada.

Si `ldd` muestra como salida `not found` para alguna biblioteca, se van a tener problemas y el programa no podra ser ejecutado. Una forma de arreglarlo es buscar la biblioteca y colocarla en el lugar correcto para que el programa loader la encuentre, que siempre busca por default en `lib` y `/usr/lib`. Si se tienen bibliotecas en otro directorio, crear una variable de ambiente `LD_LIBRARY_PATH` y poner los directorios separados por `;`.

# 1.10 Funciones de la biblioteca de UNIX

El sistema UNIX da un gran número de funciones de C. Algunas implementan operaciones de uso frecuente, mientras otras están muy especializadas en relación a su aplicación.

No reinvente la rueda. Se recomienda revisar si una función en alguna biblioteca existe, en vez de hacer la tarea de escribir su propia versión. Con lo anterior se reduce el tiempo de desarrollo de un programa. Ya que las funciones de la biblioteca han sido probadas, por lo que estarán corregidas, a diferencia de cualquiera que un programador pueda escribir. Lo anterior reducirá el tiempo de depuración del programa.

## 1.10.1 Encontrando información acerca de las bibliotecas.

El manual de UNIX tiene una entrada para todas las funciones disponibles. La documentación de funciones esta guardada en la *sección 3* del manual, y hay muchas otras útiles que hacen llamadas al sistema en la *sección 2*. Si ya sabe el nombre de la función que se quiere revisar, puede leer la página tecleando:

```
man 3 sqrt
```

Si no sabe el nombre de la función, una lista completa esta incluida en la página introductoria de la sección 3 del manual. Para leerlo, teclee:

```
man 3 intro
```

Hay aproximadamente 700 funciones. El número tiende a incrementarse con cada actualización al sistema.

En cualquier página del manual, la sección de SYNOPSIS incluye información del uso de la función. Por ejemplo:

```
#include <time.h>

char *ctime(const time_t *timep);
```

Lo que significa que se debe tener

```
#include <time.h>
```

en el archivo del programa que hace el llamado a `ctime`. Y que la función `ctime` toma un apuntador del tipo `time_t` como un argumento, y regresa una cadena `char *`.

En la sección DESCRIPTION se da una pequeña descripción de lo que hace la función.

# 1.11 Ejercicios

1. Escribe, compila y corre el siguiente programa. Modifica el programa para que incluyas tu nombre y la forma como hicistes lo anterior. Para iniciar un comentario usa `/*` y para terminarlo `*/`:

```
main()
{
    int i;

    printf("\t Numero \t\t Cubo\n\n");

    for( i=0; i<=20; ++i)
        printf("\t %d \t\t\t %d \n",i,i*i*i );
}
```

2. El siguiente programa usa la biblioteca matemática. Tecléalo, compílalo y ejecútalo. Incluye con comentarios dentro del programa tu nombre y la forma como hicistes lo anterior.

```
#include <math.h>

main()
{
    int i;

    printf("\tAngulo \t\t Seno\n\n");

    for( i=0; i<=360; i+=15)
        printf("\t %d \t\t\t %f \n",i,sin((double) i*M_PI/180.0));
}
```

3. Busca en los directorios `/lib` y `/usr/lib` las bibliotecas que están disponibles. Manda 3 nombres de estáticas y 3 de compartidas.

- Use el comando `man` para obtener detalles de las funciones de la biblioteca.
- Selecciona alguna biblioteca y ve los códigos objetos que contiene, puedes teclear:  
`ar tv biblioteca`

4. Ve en `/usr/include` que archivos de cabecera están disponibles.

- Usa los comandos `more` o `cat` para ver los archivos de texto.
- Explora los archivos cabecera para ver el contenido, observando las directivas de preprocesamiento *include*, *define*, las definiciones de los tipos y los prototipos de las funciones. Envía 10 casos donde se hayan usado las directivas anteriores, indicando de que archivo de cabecera las obtuvistes.

5. Supongamos que se tiene un programa en C el cual tiene la función principal en `main.c` y que se tienen otras funciones en los archivos `input.c` y `output.c`:

- ¿De qué forma habría que compilar y ligar este programa?
- ¿Cómo se deberán modificar los comandos anteriores para ligar una biblioteca llamada `proces01` guardada en el directorio estándar de las bibliotecas del sistema?
- ¿Cómo modificarías los comandos anteriores para ligar una biblioteca llamada `proces02` guardada en tu directorio casa?
- Algunos archivos cabecera necesitan ser leídos y encontrados en el subdirectorio

header de su directorio casa y también en directorio de trabajo actual. ¿Cómo se modificarían los comandos de compilación para hacer lo señalado?

---

---

---

---

---

## Subsecciones

- [2.1 Orígenes del C](#)
- [2.2 Características de C](#)
- [2.3 Estructura de un programa en C](#)
- [2.4 Variables](#)
  - [2.4.1 Definición de variables globales](#)
  - [2.4.2 Lectura y escritura de variables](#)
- [2.5 Constantes](#)
- [2.6 Operadores Aritméticos](#)
- [2.7 Operadores de Comparación](#)
- [2.8 Operadores lógicos](#)
- [2.9 Orden de precedencia](#)
- [2.10 Ejercicios](#)



---

## 2. Principios de C

En este capítulo se ofrece una breve historia del desarrollo del lenguaje C y se consideran también sus características.

En el resto del capítulo se ven los aspectos básicos de los programas de C, tales como su estructura, la declaración de variables, tipos de datos y operadores.

### 2.1 Orígenes del C

El proceso de desarrollo del lenguaje C se origina con la creación de un lenguaje llamado BCPL, que fue desarrollado por Martin Richards. El BCPL tuvo influencia en un lenguaje llamado B, el cual se usó en 1970 y fue inventado por Ken Thompson y que permitió el desarrollo de C en 1971, el cual lo inventó e implementó Dennis Ritchie. Para 1973 el sistema operativo UNIX estaba casi totalmente escrito en C.

Durante muchos años el estándar para C fue la versión 5 del sistema operativo UNIX, documentada en "The C Programming Language" escrito por Brian W. Kernighan and Dennis M. Ritchie in 1978 comúnmente referido como K&R.

Posteriormente se hicieron varias implementaciones las cuales mostraban las siguientes tendencias:

- Nuevas características
- Diferencias de máquinas
- Diferencias de productos
- Errores en los compiladores
- Malas implementaciones

Esto originó que en el verano de 1983 se estableciera un comité para resolver estas discrepancias, el cual empezó a trabajar en un estándar ANSI C, la cual fue completada en 1988.

## 2.2 Características de C

Algunas de las características más importantes que definen el lenguaje y que han permitido que sea tan popular, como lenguaje de programación son:

- Tamaño pequeño.
- Uso extensivo de llamadas a funciones.
- Comandos breves (poco tecleo).
- Lenguaje estructurado.
- Programación de bajo nivel (nivel bit)
- Implementación de apuntadores - uso extensivo de apuntadores para la memoria, arreglos, estructuras y funciones

Las diversas razones por la cual se ha convertido en un lenguaje de uso profesional son:

- El uso de constructores de alto nivel.
- El poder manejar actividades de bajo-nivel.
- El generar programas eficientes.
- La posibilidad de poder ser compilado en una variedad de computadoras, con pocos cambios (portabilidad).

Un punto en contra es que tiene una detección pobre de errores, lo cual en ocasiones es problemático para los principiantes.

## 2.3 Estructura de un programa en C

Un programa de C tiene básicamente la siguiente forma:

- Comandos del preprocesador.
- Definiciones de tipos.
- Prototipos de funciones - declara el tipo de función y las variables pasadas a la misma.
- Variables
- Funciones

Para un programa se debe tener una función `main()`.

Una función tiene la forma:

```
tipo nombre_de_la_funcion (parámetros)
{
    variables locales
    sentencias de C
}
```

```
}
```

Si la definición del tipo es omitida, C asume que la función regresa un tipo entero. **Nota:** Lo anterior puede ser una fuente de problemas en un programa.

A continuación se muestra un primer programa:

```
/* Programa ejemplo */  
  
main()  
{  
  
    printf( "Me gusta C\n" );  
    exit (0);  
  
}
```

NOTAS:

- C requiere un punto y coma al final de cada sentencia.
- `printf` es una función estándar de C, la cual es llamada en la función `main()`.
- `\n` significa salto de línea. Salida formateada.
- `exit()` es también una función estándar que hace que el programa termine. En el sentido estricto no es necesario ya que es la última línea de `main()` y de cualquier forma terminará el programa.

En caso de que se hubiera llamado a la función `printf` de la siguiente forma:

```
printf(".\n.1\n..2\n...3\n");
```

La salida tendría la siguiente forma:



```
.1  
..2  
...3
```

## 2.4 Variables

C tiene los siguientes tipos de datos simples:

**Tabla 2.1:** Tipos de C

<i>Tipo</i>	<i>Tamaño (bytes)</i>	<i>Límite inferior</i>	<i>Límite superior</i>
char	1	--	--
unsigned char	1	<u>0</u>	<u>255</u>
short int	2	<u>-32768</u>	<u>+32767</u>
unsigned short int	2	<u>0</u>	<u>65536</u>

(long) int	4	$-2^{31}$	$+2^{31} - 1$
float	4 	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8 	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

Los tipos de datos básicos tiene varios *modificadores* que les preceden. Se usa un modificador para alterar el significado de un tipo base para que encaje con las diversas necesidades o situaciones. Los modificadores son: signed, unsigned, long y short.

En los sistemas UNIX todos los tipos int son long int, a menos que se especifique explícitamente short int.

**Nota:** no hay un tipo booleano en C -- se deberá usar char, int o aún mejor unsigned char.

signed, unsigned, long y short pueden ser usados con los tipos char e int. Aunque es permitido el uso de signed en enteros, es redundante porque la declaración de entero por defecto asume un número con signo.

Para declarar una variable en C, se debe seguir el siguiente formato:

```
tipo lista_variables;
```

*tipo* es un tipo válido de C y *lista\_variables* puede consistir en uno o más indentificadores separados por una coma. Un identificador debe comenzar con una letra o un guión bajo.

Ejemplo:

```
int i, j, k;
float x, y, z;
char ch;
```

## 2.4.1 Definición de variables globales

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

Por ejemplo:

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

main()
{
...
}
```

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

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

main()
{
...
}
```

Que es lo mismo que:

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

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

...
}
```

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

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

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

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

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

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

```
typedef float real;
typedef char letra;

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

## 2.4.2 Lectura y escritura de variables

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

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

Por ejemplo:

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

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

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

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

Observar que se antepone & a los nombres de las variables, excepto a la cadena de caracteres. En el capítulo [8](#) que trata sobre apuntadores se revisará más a fondo el uso de este operador.

## 2.5 Constantes

ANSI C permite declarar *constantes*. Cuando se declara una constante es un poco parecido a declarar una variable, excepto que el valor no puede ser cambiado.

La palabra clave `const` se usa para declarar una constante, como se muestra a continuación:

```
const a = 1;
int a = 2;
```

### Notas:

- Se puede usar `const` antes o después del tipo.
- Es usual inicializar una constante con un valor, ya que no puede ser cambiada *de alguna otra forma*.

La directiva del preprocesador `#define` es un método más flexible para definir *constantes* en un programa.

Frecuentemente se ve la declaración `const` en los parámetros de la función. Lo anterior simplemente indica que la función no cambiara el valor del parámetro. Por ejemplo, la siguiente función usa este concepto:

```
char *strcpy(char *dest, const char *orig);
```

El segundo argumento `orig` es una cadena de C que no será alterada, cuando se use la función de la biblioteca para copiar cadenas.

## 2.6 Operadores Aritméticos

Lo mismo que en otros lenguajes de programación, en C se tienen los operadores aritméticos más usuales (+ suma, - resta, \* multiplicación, / división y % módulo).

El operador de asignación es =, por ejemplo: `i=4; ch='y';`

Incremento ++ y decremento -- unario. Los cuales son más eficientes que las respectivas asignaciones. Por ejemplo: `x++` es más rápido que `x=x+1`.

Los operadores ++ y -- pueden ser prefijos o postfijos. Cuando son prefijos, el valor es calculado antes de que la expresión sea evaluada, y cuando es postfijo el valor es calculado después que la expresión es evaluada.

En el siguiente ejemplo, ++z es prefijo y -- es postfijo:

```
int x, y, z;

main()
{
    x=( ( ++z ) - ( y-- ) ) % 100;
}
```

Que es equivalente a:

```
int x, y, z;

main()
{
    z++;
    x = ( z-y ) % 100;
    y--;
}
```

El operador % (módulo o residuo) solamente trabaja con enteros, aunque existe una función para flotantes ([15.1](#) `fmod()`) de la biblioteca matemática.

El operador división / es para división entera y flotantes. Por lo tanto hay que tener cuidado. El resultado de `x = 3 / 2;` es uno, aún si x es declarado como float. La regla es: si ambos argumentos en una división son enteros, entonces el resultado es entero. Si se desea obtener la división con la fracción, entonces escribirlo como: `x = 3.0 / 2;` o `x = 3 / 2.0` y aún mejor `x = 3.0 / 2.0`.

Por otra parte, existe una forma más corta para expresar cálculos en C. Por ejemplo, si se tienen expresiones como: `i = i + 3;` o `x = x * (y + 2);`, pueden ser reescritas como:

`r1 oper = expr2`

Lo cual es equivalente, pero menos eficiente que:

`r1 = expr1 oper expr2`

Por lo que podemos reescribir las expresiones anteriores como: `i += 3; y x *= y + 2;` respectivamente.

## 2.7 Operadores de Comparación

El operador para probar la igualdad es `==`, por lo que se deberá tener cuidado de no escribir accidentalmente sólo `=`, ya que:

```
if ( i = j ) ...
```

Es una sentencia legal de C (sintácticamente hablando aunque el compilador avisa cuando se emplea), la cual copia el valor de `j` en `i`, lo cual será interpretado como VERDADERO, si `j` es diferente de cero.

Diferente es `!=`, otros operadores son: `<` menor que, `>` mayor que, `<=` menor que o igual a y `>=` (mayor que o igual a).

## 2.8 Operadores lógicos

Los operadores lógicos son usualmente usados con sentencias condicionales o relacionales, los operadores básicos lógicos son:

`&&` Y lógico, `||` O lógico y `!` negación.

## 2.9 Orden de precedencia

Es necesario ser cuidadosos con el significado de expresiones tales como `a + b * c`, dependiendo de lo que se desee hacer

```
(a + b) * c
```

o

```
a + (b * c)
```

Todos los operadores tienen una prioridad, los operadores de mayor prioridad son evaluados antes que los que tienen menor prioridad. Los operadores que tienen la misma prioridad son evaluados de izquierda a derecha, por lo que:

```
a - b - c
```

es evaluado como

```
(a - b) - c
```

<i>Prioridad</i>	<i>Operador(es)</i>
Más alta	( ) [ ] ->
	! ~ ++ -- - (tipo) * & sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	&&
	?
	= += -= *= /=
Más baja	,

De acuerdo a lo anterior, la siguiente expresión:

```
a < 10 && 2 * b < c
```

Es interpretada como:

```
( a < 10 ) && ( ( 2 * b ) < c )
```

y

```
a =
b = 10 / 5
    + 2;
```

como

```
a =
( b =
  ( 10 / 5 )
    + 2 );
```

## 2.10 Ejercicios

Escribir programas en C para hacer las siguientes tareas:

1. Leer la entrada de dos números y mostrar el doble producto del primero menos la mitad del segundo.
  2. Lea y escriba su nombre, apellido paterno, apellido materno y matricula en un formato adecuado.
  3. Escribir un programa para leer un ``flotante" que representa un número de grados Celsius, e imprime como un ``flotante" la temperatura equivalente en grados Fahrenheit. La salida puede ser de la siguiente forma: 100.0 grados Celsius son 212.0 grados Fahrenheit.
  4. Escribir un programa para imprimir varias veces el ejercicio 2. Puede usar varias instrucciones printf, con un caracter de nueva línea en cada una, o una instrucción con varios caracteres nueva línea en la cadena de formateo.
  5. Escribir un programa que lea el radio de un círculo como un número flotante y muestre el área y el perímetro del círculo.
  6. Dados ciertos centímetros como entrada de tipo flotante, imprimir su equivalencia a pies (enteros) y pulgadas (flotante, 1 decimal), dando las pulgadas con una precisión de un lugar decimal. Suponer 2.54 centímetros por pulgada, y 12 pulgadas por pie.  
Si la entrada es 333.3, el formato de la salida deberá ser:  
333.3 centímetros son 10 pies 11.2 pulgadas.
- 
-

## Subsecciones

- [3.1 La sentencia `if`](#)
  - [3.2 El operador `?`](#)
  - [3.3 La sentencia `switch`](#)
  - [3.4 Ejercicios](#)
- 

# 3. Estructuras Condicionales

En este capítulo se revisan los distintos métodos con los que C controla el **flujo** lógico de un programa.

Como se revisó en el capítulo anterior, los operadores relaciones binarios que se usan son:

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

además los operadores lógicos binarios:

`||, &&`

y el operador lógico unario de negación `!`, que sólo toma un argumento.

Los operadores anterior son usados con las siguientes estructuras que se muestran.

## 3.1 La sentencia `if`

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

```
if (condicion)
    sentencia;
...0
```

```
if (condicion)
    sentencia1;
else
    sentencia2;
...0
```

```

if (condicion1)

    sentencia1;

else if (condicion2)

    sentencia2;

...
else

    sentencian;

```

El flujo lógico de esta estructura es de arriba hacia abajo. La primera sentencia se ejecutará y se saldrá de la estructura `if` si la primera condición es verdadera. Si la primera condición fue falsa, y existe otra condición, se evalúa, y si la condición es verdadera, entonces se ejecuta la sentencia asociada. Si existen más condiciones dentro de la estructura `if`, se van evaluando éstas, siempre y cuando las condiciones que le precedan sean falsas.

La sentencia que esta asociada a la palabra reservada `else`, se ejecuta si todas las condiciones de la estructura `if` fueron falsas.

Por ejemplo:

```

main()
{
    int x, y, w;

        .....

    if (x>0)
    {
        z=w;
        .....
    }
    else
    {
        z=y;
        .....
    }
}

```

## 3.2 El operador ?

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

```

expresion1 ? expresion2 : expresion3;

```

Que es equivalente a la siguiente expresión:

```

if (expresion1) then expresion2 else expresion3;

```

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

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

que es lo mismo que:

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

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

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

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

main()
{
    int t;

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

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

## 3.3 La sentencia `switch`

Aunque con la estructura `if ... else if` se pueden realizar comprobaciones múltiples, en ocasiones no es muy elegante, ya que el código puede ser difícil de seguir y puede confundir incluso al autor transcurrido un tiempo. Por lo anterior, C tiene incorporada una sentencia de bifurcación múltiple llamada `switch`. Con esta sentencia, la computadora comprueba una variable sucesivamente frente a una lista de constantes enteras o de carácter. Después de encontrar una coincidencia, la computadora ejecuta la sentencia o bloque de sentencias que se asocian con la constante. La forma general de la sentencia `switch` es:

```
switch (variable) {

    case constant1:

        secuencia de sentencias
        break;
```

```

    case constante2:
        secuencia de sentencias
        break;

    case constante3:
        secuencia de sentencias
        break;

    ...

    default:
        secuencia de sentencias

}

```

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

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

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

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

```

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

    case ' ':
        numesp++;
        break;

    default:
        numotras++;
        break;
}

```

## 3.4 Ejercicios

1. Escribir un programa que lea dos caracteres, e imprima su valor cuando se pueda interpretar como un número hexadecimal. Aceptar letras mayúsculas y minúsculas para los valores del 10 al 15.
  2. Leer un valor entero. Suponer que el número es un día de la semana. Suponer que 0 corresponde a Domingo y así sucesivamente. Imprimir el nombre del día.
  3. Dados como entrada 3 enteros representando la fecha como día, mes, año, imprimir la fecha del día anterior. Por ejemplo para una entrada como: 1 3 1992 La salida será: Fecha anterior a 1-3-1992 es 29-02-1992
  4. Escribir un programa el cual lea dos valores enteros. Si el primero es menor que el segundo, que imprima el mensaje ``Arriba''. Si el segundo es menor que el primero, que imprima el mensaje ``Abajo''. Si los números son iguales, que imprima el mensaje ``igual''. Si hay un error en la lectura de los datos, que imprima un mensaje conteniendo la palabra ``Error" y haga `exit( 0 );`
- 
-

## Subsecciones

- [4.1 La sentencia for](#)
  - [4.2 La sentencia while](#)
  - [4.3 La sentencia do-while](#)
  - [4.4 Uso de break y continue](#)
  - [4.5 Ejercicios](#)
- 

# 4. Iteración

En este capítulo se revisan los mecanismos de C para repetir un conjunto de instrucciones hasta que se cumple cierta condición.

## 4.1 La sentencia for

La sentencia `for` tiene el siguiente formato:

```
for ( expresion1; expresion2; expresion3)  
  
    sentencia;  
    o { bloque de sentencias }
```

En donde *expresion1* se usa para realizar la *inicialización* de variables, usando una o varias sentencias, si se usan varias sentencias deberá usarse el operador `,` para separarlas. Por lo general, establece el valor de la variable de control del ciclo. *expresion2* se usa para la condición de terminación del ciclo y *expresion3* es el modificador a la variable de control del ciclo cada vez que la computadora lo repite, pero también puede ser más que un incremento.

Por ejemplo:

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

genera la siguiente salida a pantalla ...

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

Todos las siguientes sentencias `for` son válidas en C. Las aplicaciones prácticas de tales sentencias

no son importantes aquí, ya que tan sólo se intenta ilustrar algunas características que pueden ser de utilidad:

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

En el segundo ejemplo se muestra la forma como múltiples expresiones pueden aparecer, siempre y cuando estén separadas por una coma ,

En el tercer ejemplo, el ciclo continuará iterando hasta que *z* se convierta en 0.

## 4.2 La sentencia `while`

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

```
while ( expresion ) sentencia;
```

donde *sentencia* puede ser una sentencia vacía, una sentencia única o un bloque de sentencias que se repetirán. Cuando el flujo del programa llega a esta instrucción, primero se revisa si la condición es verdad para ejecutar la(s) sentencia(s), y después el ciclo `while` se repetirá mientras la *condición* sea verdadera. Cuando llega a ser falsa, el control del programa pasa a la línea que sigue al ciclo.

En el siguiente ejemplo se muestra una rutina de entrada desde el teclado, la cual se cicla mientras no se pulse A:

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

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

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

A continuación mostramos otro ejemplo:

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

```

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

```

que genera la siguiente salida en pantalla:

```

x = 3
x = 2
x = 1

```

Como se observa, dentro del ciclo tenemos más de una sentencia, por lo que se requiere usar la llave abierta y la llave cerrada { . . . } para que el grupo de sentencias sean tratadas como una unidad.

Como el ciclo `while` pueda aceptar también expresiones, y no solamente condiciones lo siguiente es válido:

```

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

```

Si se usan este tipo de expresiones, solamente cuando el resultado de `x--`, `x=x+1` o `x+=5` sea cero, la condición fallará y se podrá salir del ciclo.

De acuerdo a lo anterior, podemos realizar una operación completa dentro de la expresión. Por ejemplo:

```

main()
{
    char carac;

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

```

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

## 4.3 La sentencia `do-while`

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

```

do {

    sentencia;
}

```

```
    } while (condición);
```

Aunque no son necesarias las llaves cuando sólo está presente una sentencia, se usan normalmente por legibilidad y para evitar confusión (respecto al lector, y no del compilador) con la sentencia `while`.

En el siguiente programa se usa un ciclo `do ... while` para leer números desde el teclado hasta que uno de ellos es menor que o igual a 100:

```
main()
{
    int num;

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

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

```
main()
{
    int opc;

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

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

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

Se muestra un ejemplo donde se reescribe usando `do ... while` uno de los ejemplos ya mostrados.

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

    do
```

```

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

```

## 4.4 Uso de break y continue

Como se comento uno de los usos de la sentencia `break` es terminar un `case` en la sentencia `switch`. Otro uso es forzar la terminación inmediata de un ciclo, saltando la prueba condicional del ciclo.

Cuando se encuentra la sentencia `break` en un bucle, la computadora termina inmediatamente el ciclo y el control del programa pasa a la siguiente sentencia del ciclo. Por ejemplo:

```

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

```

Este programa muestra en pantalla los números del 0 al 10, cuando alcanza el valor 10 se cumple la condición de la sentencia `if`, se ejecuta la sentencia `break` y sale del ciclo.

La sentencia `continue` funciona de manera similar a la sentencia `break`. Sin embargo, en vez de forzar la salida, `continue` fuerza la siguiente iteración, por lo que salta el código que falta para llegar a probar la condición. Por ejemplo, el siguiente programa visualizará sólo los números pares:

```

main()
{
    int x;

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

```

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

```

main()
{
    int valor;

    while( scanf("%d", &valor) == 1 && valor != 0)
    {
        if ( valor<0 )

```

```

    {
        printf("Valor no valido\n");
        break;
        /* Salir del ciclo */
    }

    if ( valor>100)
    {
        printf("Valor no valido\n");
        continue;
        /* Pasar al principio del ciclo nuevamente */
    }

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

```

## 4.5 Ejercicios

1. Escribir un programa que lea 5 números y encuentre el promedio, el máximo y el mínimo de esos valores.
2. Escribir un programa que lea números hasta que se encuentre el cero. El segundo número se sumará al primero, luego el tercero se restará, el cuarto se sumará, y así se deberá seguir alternado hasta que se llegue al cero. Cuando se llegue a esta condición deberá imprimir el resultado, el total de operandos de la operación (sin incluir el cero), y la suma de los operandos que se restaron.
3. Escribir un programa que lea un valor entero que será la base para un sistema numérico (binario, octal o decimal), después que lea un entero positivo en esa base y que imprima su valor en base 10. Se debe validar que el número pertenezca a esa base. La base será menor que o igual a 10. El programa podría tener la siguiente salida:

Entrada		Salida
Base	Numero	
10	1234	1234
8	77	63
2	1111	15

4. Escribir un programa que lea un número en base 10 y lo convierta a base 2, base 8 y base hexadecimal.
5. Leer tres valores representando lo siguiente:
  - El capital (número entero de pesos)
  - Una tasa de interés en porcentaje (flotante)
  - y un número de años (entero).

Calcular los valores de la suma del capital y el interés compuesto para un período dado de años. Para cada año el interés es calculado como: `interes = capital * tasa_interes / 100;`

el cual se suma al capital

```
capital += interes;
```

Imprimir los valores de moneda con una precisión de dos decimales. Imprimir los valores del interés compuesto para cada año al final del período. La salida puede ser como la siguiente:

Capital inicial 35000.00 con tasa del 12.50 en 10 años

Año	Interes	Suma
1	4375.00	39375.00
2	4921.88	44296.88
3	5537.11	49833.98
4	6229.25	56063.23
5	7007.90	63071.14
6	7883.89	70955.03
7	8869.38	79824.41
8	9978.05	89802.45
9	11225.31	101027.76
10	12628.47	113656.23

6. Leer un valor positivo, y hacer la siguiente secuencia: si el número es par, dividirlo entre 2; si es non, multiplicarlo por 3 y sumarle 1. Repetir lo anterior hasta que el valor sea 1, imprimiendo cada valor, también se deberá imprimir cuantas operaciones de estas son hechas.

Una salida podría ser la siguiente:

```
El valor inicial es 9
El siguiente valor es 28
El siguiente valor es 14
El siguiente valor es 7
El siguiente valor es 22
El siguiente valor es 11
El siguiente valor es 34
El siguiente valor es 17
El siguiente valor es 52
El siguiente valor es 26
El siguiente valor es 13
El siguiente valor es 40
El siguiente valor es 20
El siguiente valor es 10
El siguiente valor es 5
El siguiente valor es 16
El siguiente valor es 8
El siguiente valor es 4
El siguiente valor es 2
Valor final 1, numero de pasos 19.
```

Si el valor ingresado es menor que 1, imprimir un mensaje que contenga la palabra

Error

y haga

```
exit(0)
```

---

---

---

## Subsecciones

- [5.1 Arreglos unidimensionales y multidimensionales](#)
  - [5.2 Cadenas](#)
  - [5.3 Ejercicios](#)
- 

# 5. Arreglos y cadenas

En el siguiente capítulo se presentan los arreglos y las cadenas. Las cadenas se consideran como un arreglo de tipo *char*.

## 5.1 Arreglos unidimensionales y multidimensionales

Los arreglos son una colección de variables del mismo tipo que se referencian utilizando un nombre común. Un arreglo consta de posiciones de memoria contigua. La dirección más baja corresponde al primer elemento y la más alta al último. Un arreglo puede tener una o varias dimensiones. Para acceder a un elemento en particular de un arreglo se usa un índice.

El formato para declarar un arreglo unidimensional es:

```
tipo nombre_arr [ tamaño ]
```

Por ejemplo, para declarar un arreglo de enteros llamado *listanum* con diez elementos se hace de la siguiente forma:

```
int listanum[10];
```

En C, todos los arreglos usan cero como índice para el primer elemento. Por tanto, el ejemplo anterior declara un arreglo de enteros con diez elementos desde **listanum[0]** hasta **listanum[9]**.

La forma como pueden ser accedados los elementos de un arreglo, es de la siguiente forma:

```
listanum[2] = 15; /* Asigna 15 al 3er elemento del arreglo listanum*/  
num = listanum[2]; /* Asigna el contenido del 3er elemento a la variable num */
```

El lenguaje C no realiza comprobación de contornos en los arreglos. En el caso de que sobrepase el final durante una operación de asignación, entonces se asignarán valores a otra variable o a un trozo del código, esto es, si se dimensiona un arreglo de tamaño *N*, se puede referenciar el arreglo por encima de *N* sin provocar ningún mensaje de error en tiempo de compilación o ejecución, incluso

aunque probablemente se provoque el fallo del programa. Como programador se es responsable de asegurar que todos los arreglos sean lo suficientemente grandes para guardar lo que pondrá en ellos el programa.

C permite arreglos con más de una dimensión , el formato general es:

```
tipo nombre_arr [ tam1 ] [ tam2 ] ... [ tamN ] ;
```

Por ejemplo un arreglo de enteros bidimensionales se escribirá como:

```
int tabladenums[50][50];
```

Observar que para declarar cada dimensión lleva sus propios paréntesis cuadrados.

Para acceder los elementos se procede de forma similar al ejemplo del arreglo unidimensional, esto es,

```
tabladenums[2][3] = 15; /* Asigna 15 al elemento de la 3ª fila y la 4ª columna*/
num = tabladenums[25][16];
```

A continuación se muestra un ejemplo que asigna al primer elemento de un arreglo bidimensional cero, al siguiente 1, y así sucesivamente.

```
main()
{
    int t,i,num[3][4];

    for(t=0; t<3; ++t)
        for(i=0; i<4; ++i)
            num[t][i]=(t*4)+i*1;

    for(t=0; t<3; ++t)
    {
        for(i=0; i<4; ++i)
            printf("num[%d][%d]=%d  ", t,i,num[t][i]);
        printf("\n");
    }
}
```

En C se permite la inicialización de arreglos, debiendo seguir el siguiente formato:

```
tipo nombre_arr[ tam1 ] [ tam2 ] ... [ tamN ] = {lista-valores};
```

Por ejemplo:

```
int i[10] = {1,2,3,4,5,6,7,8,9,10};
int num[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};
```

## 5.2 Cadenas

A diferencia de otros lenguajes de programación que emplean un tipo denominado cadena *string* para manipular un conjunto de símbolos, en C, se debe simular mediante un arreglo de caracteres,

en donde la terminación de la cadena se debe indicar con nulo. Un nulo se especifica como `'\0'`. Por lo anterior, cuando se declare un arreglo de caracteres se debe considerar un carácter adicional a la cadena más larga que se vaya a guardar. Por ejemplo, si se quiere declarar un arreglo `cadena` que guarde una cadena de diez caracteres, se hará como:

```
char cadena[11];
```

Se pueden hacer también inicializaciones de arreglos de caracteres en donde automáticamente C asigna el carácter nulo al final de la cadena, de la siguiente forma:

```
char nombre_arr[ tam ]="cadena";
```

Por ejemplo, el siguiente fragmento inicializa `cadena` con ```hola```:

```
char cadena[5]="hola";
```

El código anterior es equivalente a:

```
char cadena[5]={'h','o','l','a','\0'};
```

Para asignar la entrada estándar a una cadena se puede usar la función `scanf` con la opción `%s` (observar que no se requiere usar el operador `&`), de igual forma para mostrarlo en la salida estándar.

Por ejemplo:

```
main()
{
    char nombre[15], apellidos[30];

    printf("Introduce tu nombre: ");
    scanf("%s", nombre);
    printf("Introduce tus apellidos: ");
    scanf("%s", apellidos);
    printf("Usted es %s %s\n", nombre, apellidos);
}
```

El lenguaje C no maneja cadenas de caracteres, como se hace con enteros o flotantes, por lo que lo siguiente no es válido:

```
main()
{
    char nombre[40], apellidos[40], completo[80];

    nombre="José María";           /* Ilegal */
    apellidos="Morelos y Pavón";   /* Ilegal */
    completo="Gral."+nombre+apellidos; /* Ilegal */
}
```

## 5.3 Ejercicios

1. Escribir un programa que lea un arreglo de cualquier tipo (entero, flotante, char), se podría pedir al usuario que indique el tipo de arreglo, y también escribir un programa que revise el arreglo para encontrar un valor en particular.

2. Leer un texto, un caracter a la vez desde la entrada estándar (que es el teclado), e imprimir cada línea en forma invertida. Leer hasta que se encuentre un *final-de-datos* (teclar CONTROL-D para generarlo).

El programa podría probarse tecleando `progreV | progrev` para ver si una copia exacta de la entrada original es recreada.

Para leer caracteres hasta el final de datos, se puede usar un ciclo como el siguiente

```
char ch;
while( ch=getchar(), ch>=0 ) /* ch < 0 indica fin-de-datos */
```

o

```
char ch;
while( scanf( "%c", &ch ) == 1 ) /* se lee un caracter */
```

3. Escribir un programa para leer un texto hasta el fin-de-datos, y mostrar una estadística de las longitudes de las palabras, esto es, el número total de palabras de longitud 1 que hayan ocurrido, el total de longitud 2 y así sucesivamente.

Define una palabra como una secuencia de caracteres alfabéticos. Se deberán permitir palabras hasta de una longitud de 25 letras.

Una salida típica podría ser como esta:

```
longitud  1 : 16 ocurrencias
longitud  2 : 20 ocurrencias
longitud  3 :  5 ocurrencias
longitud  4 :  2 ocurrencias
longitud  5 :  0 ocurrencias
.....
```

---

---

---

---

## Subsecciones

- [6.1 Funciones void](#)
  - [6.2 Funciones y arreglos](#)
  - [6.3 Prototipos de funciones](#)
  - [6.4 Ejercicios](#)
- 

# 6. Funciones

Una función es un conjunto de declaraciones, definiciones, expresiones y sentencias que realizan una tarea específica.

El formato general de una función en C es

```
especificador_de_tipo nombre_de_función( lista_de_parámetros )
{
    variables locales
    código de la función
}
```

El *especificador\_de\_tipo* indica el tipo del valor que la función devolverá mediante el uso de `return`. El valor puede ser de cualquier tipo válido. Si no se especifica un valor, entonces la computadora asume por defecto que la función devolverá un resultado entero. No se tienen siempre que incluir parámetros en una función. la lista de parámetros puede estar vacía.

Las funciones terminan y regresan automáticamente al procedimiento que las llamó cuando se encuentra la última llave `}`, o bien, se puede forzar el regreso antes usando la sentencia `return`. Además del uso señalado la función `return` se usa para devolver un valor.

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

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

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

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

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

## 6.1 Funciones void

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

```
void cuadrados()
{
    int contador;

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

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

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

## 6.2 Funciones y arreglos

Cuando se usan un arreglo como un argumento a la función, se pasa sólo la dirección del arreglo y no la copia del arreglo entero. Para fines prácticos podemos considerar el nombre del arreglo sin ningún índice como la dirección del arreglo.

Considerar el siguiente ejemplo en donde se pasa un arreglo a la función `imp_rev`, observar que no es necesario especificar la dimensión del arreglo cuando es un parámetro de la función.

```
void imp_rev(char s[])
{
    int t;

    for( t=strlen(s)-1; t>=0; t--)
        printf("%c", s[t]);
}

main()
{
    char nombre[]="Facultad";

    imp_rev(nombre);
}
```

Observar que en la función `imp_rev` se usa la función `strlen` para calcular la longitud de la cadena sin incluir el terminador nulo. Por otra parte, la función `imp_rev` no usa la sentencia `return` ni para terminar de usar la función, ni para regresar algún valor.

Se muestra otro ejemplo,

```
float enconprom(int tam, float lista[])
{
    int i;
    float suma = 0.0;

    for ( i=0; i<tam; i++)
        suma += lista[i];
    return(suma/tam);
}

main()
{
    float numeros[]={2.3, 8.0, 15.0, 20.2, 44.01, -3.0, -2.9};

    printf("El promedio de la lista es %f\n", enconprom(7,numeros) );
}
```

Para el caso de que se tenga que pasar un arreglo con más de una dimensión, no se indica la primera dimensión pero, el resto de las dimensiones deben señalarse. Se muestra a continuación un ejemplo:

```
void imprtabla(int tamx,int tamy, float tabla[][5])
{
    int x,y;

    for ( x=0; x<tamx; x++ )
    {
        for ( y=0; y<tamy; y++ )
            printf("t[%d][%d]=%f", x,y,tabla[x][y]);
        printf("\n");
    }
}
```

## 6.3 Prototipos de funciones

Antes de usar una función C debe tener *conocimiento* acerca del tipo de dato que regresará y el tipo de los parámetros que la función espera.

El estándar ANSI de C introdujo una nueva (mejor) forma de hacer lo anterior respecto a las versiones previas de C.

La importancia de usar prototipos de funciones es la siguiente:

- Se hace el código más estructurado y por lo tanto, más fácil de leer.
- Se permite al compilador de C revisar la *sintaxis* de las funciones llamadas.

Lo anterior es hecho, dependiendo del alcance de la función. Básicamente si una función ha sido definida antes de que sea usada (o llamada), entonces se puede usar la función sin problemas.

Si no es así, entonces la función se debe *declarar*. La declaración simplemente maneja el tipo de dato que la función regresa y el tipo de parámetros usados por la función.

Es una práctica usual y conveniente escribir el prototipo de todas las funciones al principio del programa, sin embargo esto no es estrictamente necesario.

Para *declarar* un prototipo de una función se indicará el tipo de dato que regresará la función, el nombre de la función y entre paréntesis la lista del tipo de los parámetros de acuerdo al orden que aparecen en la definición de la función. Por ejemplo:

```
int longcad(char []);
```

Lo anterior declara una función llamada `longcad` que regresa un valor entero y acepta una cadena como parámetro.

## 6.4 Ejercicios

1. Escribir una función `reemplaza`, la cual toma una cadena como parámetro, le reemplaza todos los espacios de la cadena por un guión bajo, y devuelve el número de espacios reemplazados. Por ejemplo:

```
char cadena[] = "El gato negro";  
n = reemplaza( cadena );
```

deberá devolver:

```
cadena convertida "El_gato_negro"  
n = 2
```

2. Escribir un programa que lea una línea de texto en un buffer (una cadena de caracteres) usando la función `gets` y calcule la longitud de la línea (NO usar la función `strlen`).
  3. Modificar el programa anterior para que lea un archivo de texto. El archivo deberá redireccionarse al programa, debiendo mostrar el contenido del mismo. En caso de que se lea una línea con longitud 0 deberá terminar el programa.
- 
-

---

## Subsecciones

- [7.1 Estructuras](#)
    - [7.1.1 Definición de nuevos tipos de datos](#)
  - [7.2 Uniones](#)
  - [7.3 Conversión de tipos \(casts\)](#)
  - [7.4 Enumeraciones](#)
  - [7.5 Variables estáticas](#)
  - [7.6 Ejercicios](#)
- 

# 7. Más tipos de datos

En este capítulo se revisa la forma como pueden ser creados y usados en C tipos de datos más complejos y estructuras.

## 7.1 Estructuras

En C una estructura es una colección de variables que se referencian bajo el mismo nombre. Una estructura proporciona un medio conveniente para mantener junta información que se relaciona. Una *definición de estructura* forma una plantilla que se puede usar para crear variables de estructura. Las variables que forman la estructura son llamados *elementos estructurados*.

Generalmente, todos los elementos en la estructura están relacionados lógicamente unos con otros. Por ejemplo, se puede representar una lista de nombres de correo en una estructura. Mediante la palabra clave `struct` se le indica al compilador que defina una plantilla de estructura.

```
struct direc
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
};
```

Con el trozo de código anterior *no ha sido declarada ninguna variable*, tan sólo se ha definido el formato. Para declarar una variable, se hará como sigue:

```
struct direc info_direc;
```

Se pueden declarar una o más variables cuando se define una estructura entre `)` y `;`. Por ejemplo:

```
struct direc
{
    char nombre[30];
```

```

char calle[40];
char ciudad[20];
char estado[3];
unsigned int codigo;
} info_direc, binfo, cinfo;

```

observar que `direc` es una *etiqueta* para la estructura que sirve como una forma breve para futuras declaraciones. Como en esta última declaración se indican las variables con esta estructura, se puede omitir el nombre de la estructura tipo.

Las estructuras pueden ser también preinicializadas en la declaración:

```

struct direc info_direc={"Vicente Fernandez", "Fantasia
2000", "Dorado", "MMX", 12345};

```

Para referenciar o acceder un miembro (o campo) de una estructura, C proporciona el operador punto `.`, por ejemplo, para asignar a `info_direc` otro código, lo hacemos como:

```

info_direc.codigo=54321;

```

## 7.1.1 Definición de nuevos tipos de datos

Se señaló previamente (sección [2.4.1](#)) que `typedef` se puede usar para definir nuevos nombres de datos explícitamente, usando algunos de los tipos de datos de C, donde su formato es:

```

typedef <tipo> <nombre>;

```

Se puede usar `typedef` para crear nombres para tipos más complejos, como una estructura, por ejemplo:

```

typedef struct direc
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
} sdirec;

sdirec info_direc={"Vicente Fernandez", "Fantasia 2000", "Dorado", "MMX", 12345};

```

en este caso `direc` sirve como una *etiqueta* a la estructura y es opcional, ya que ha sido definido un nuevo tipo de dato, por lo que la etiqueta no tiene mucho uso, en donde `sdirec` es el nuevo tipo de datos e `info_direc` es una variable del tipo `sdirec`, la cual es una estructura.

Con C también se pueden tener arreglos de estructuras:

```

typedef struct direc
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
}

```

```
} info_direc;

info_direc artistas[1000];
```

por lo anterior, `artistas` tiene 1000 elementos del tipo `info_direc`. Lo anterior podría ser accesado de la siguiente forma:

```
artistas[50].codigo=22222;
```

## 7.2 Uniones

Una union es una variable la cual podría guardar (en momentos diferentes) objetos de diferentes tamaños y tipos. C emplea la sentencia `union` para crear uniones por ejemplo:

```
union numero
{
    short  shortnumero;
    long   longnumero;
    double floatnumero;
} unnumero;
```

con lo anterior se define una unión llamada `numero` y una instancia de esta llamada `unnumero`. `numero` es la etiqueta de la unión y tiene el mismo comportamiento que la etiqueta en la estructura.

Los miembros pueden ser accesados de la siguiente forma:

```
printf("%ld\n", unnumero.longnumero);
```

con la llamada a la función se muestra el valor de `longnumero`.

Cuando el compilador de C esta reservando memoria para las uniones, siempre creará una variable lo suficientemente grande para que quepa el tipo de variable más largo de la unión.

Con la finalidad de que el programa pueda llevar el registro del tipo de la variable unión usada en un momento dado, es común tener una estructura (con una unión anidada) y una variable que indica el tipo de unión.

Se muestra un ejemplo de lo anterior:

```
typedef struct
{
    int maxpasajeros;
} jet;

typedef struct
{
    int capac_elev;
} helicoptero;

typedef struct
{
    int maxcarga;
} avioncarga;
```

```

typedef union
{
    jet jetu;
    helicoptero helicoptero;
    avioncarga avioncargau;
} transporteaereo;

typedef struct
{
    int tipo;
    int velocidad;
    transporteaereo descripcion;
} un_transporteaereo

```

en el ejemplo se define una unión base de transporte aéreo el cual puede ser un jet, un helicóptero o un avion de carga.

En la estructura `un_transporeaereo` hay un miembro para el tipo, que indica cual es la estructura manejada en ése momento.

## 7.3 Conversión de tipos (casts)

C es uno de los pocos lenguajes que permiten la conversión de tipos, esto es, forzar una variable de un tipo a ser de otro tipo. Lo anterior se presenta cuando variables de un tipo se mezclan con las de otro tipo. Para llevar a cabo lo anterior se usa el operador de conversión (cast) `()`. Por ejemplo:

```

int numeroentero;
float numeroflotante = 9.87;

numeroentero = (int) numeroflotante;

```

con lo cual se asigna 9 a la variable `numeroentero` y la fracción es desechada.

El siguiente código:

```

int numeroentero = 10;
float numeroflotante;

numeroflotante = (float) numeroentero;

```

asigna `10.0` a `numeroflotante`. Como se observa C convierte el valor del lado derecho de la asignación al tipo del lado izquierdo.

La conversión de tipos puede ser también usada con cualquier tipo simple de datos incluyendo `char`, por lo tanto:

```

int numeroentero;
char letra = 'A';

numeroentero = (int) letra;

```

asigna 65 (que es el código ASCII de 'A') a `numeroentero`.

Algunas conversiones de tipos son hechas automáticamente - esto es principalmente por la característica de compatibilidad de tipos.

Una buena regla es la siguiente: *En caso de duda, conversión de tipos.*

Otro uso es asegurarse que la división de números se comporta como se requiere, ya que si se tienen dos enteros la forma de forzar el resultado a un número flotante es:

```
numeroflotante = (float) numerent / (float) denoment;
```

con lo que se asegura que la división devolverá un número flotante.

## 7.4 Enumeraciones

Una *enumeración* es un conjunto de constantes enteras con nombre y especifica todos los valores legales que puede tener una variable del tipo *enum*.

La forma como se define una enumeración es de forma parecida a como se hace con las estructuras, usando la palabra clave `enum` para el comienzo de un tipo de enumeración. Su formato es:

```
enum nombre_enum { lista_de_enumeración } lista_de_variables;
```

Es opcional **nombre\_enum** y **lista\_de\_variables**. La primera se usa para declarar las variables de su tipo. El siguiente fragmento define una enumeración llamada `disco` que declara `almacenamiento` para ser de ese tipo.

```
enum almacenamiento { diskette, dd, cd, dvd, cinta };
```

```
enum almacenamiento disco;
```

Con la siguiente definición y declaración son válidas las siguientes sentencias:

```
disco = cd;
```

```
if ( disco == diskette )  
    printf("Es de 1440 Kb\n");
```

Se inicializa el primer símbolo de enumeración a cero, el valor del segundo símbolo a 1 y así sucesivamente, a menos que se inicialice de otra manera. Por tanto,

```
printf("%d %d\n", dd, cinta)
```

muestra 1 4 en la pantalla.

Se puede especificar el valor de uno o más símbolos usando un inicializador. Para hacerlo, poner un signo igual y un valor entero después del símbolo.

Por ejemplo, lo siguiente asigna el valor 250 a `cd`

```
enum disco { diskette, duro, cd=250, dvd, cinta };
```

Por lo tanto los valores de los símbolos son los siguientes:

<b>diskette</b>	0
<b>duro</b>	1
<b>cd</b>	250
<b>dvd</b>	251
<b>cinta</b>	252

## 7.5 Variables estáticas

C soporta cuatro especificadores de clase de almacenamiento. Son

**auto**  
**extern**  
**static**  
**register**

Estos especificadores le dicen al compilador como almacenar la variable que sigue. El especificador de almacenamiento precede a la declaración de variable que tiene el formato general:

```
especif_almac especific_tipo lista_variables;
```

Se usa el especificador **auto** para declarar variables locales. Sin embargo, raramente se usa porque las variables locales son **auto** por defecto.

Las variables **static** son variables permanentes en su propia función o archivo. Se diferencian de las variables globales porque son desconocidas fuera de sus funciones o archivo, pero mantienen sus valores entre llamadas. Esta característica las hace útiles cuando se escriben funciones generales y bibliotecas de funciones.

**Variables static locales.** Una variable estática local es una variable local que retienen su valor entre llamadas de función, ya que C les crea un almacenamiento permanente. Si lo anterior no se pudiera hacer, entonces se tendrían que usar variables globales -que abriría la puerta a posibles efectos laterales-. Un ejemplo de una función que requeriría tales variables es un generador de series de números que produce un número nuevo basado en el último.

A continuación se muestra un ejemplo en donde se analiza el comportamiento de una variable **auto** y una variable **static**

```
void stat(); /* Prototipo de la funcion */

main()
{
    int i;
    for (i=0; i<5; ++i)
        stat();
}
```

```
void stat()
{
    auto int a_var = 0;
    static int s_var = 0;

    printf("auto = %d, static = %d \n", a_var, s_var);
    ++a_var;
    ++s_var;
}
```

La salida del código anterior es:

```
auto = 0, static = 0
auto = 0, static = 1
auto = 0, static = 2
auto = 0, static = 3
auto = 0, static = 4
```

Como se puede observar la variable `a_var` es creada cada vez que se llama a la función. La variable `s_var` es creada en la primera llamada y después recuerda su valor, es decir, no se destruye cuando termina la función.

## 7.6 Ejercicios

1. Escribir un programa que use el tipo `enum` para mostrar el nombre de un mes, su predecesor y su sucesor. El mes se ingresará desde el teclado dando un número entre 1 y 12.
  2. Escribir un programa que contenga una función que pueda recibir dos estructuras que contienen las coordenadas en el plano de dos puntos dados, de los cuales se desea conocer su punto medio.
- 
-

---

## Subsecciones

- [8.1 Definición de un apuntador](#)
  - [8.2 Apuntadores y Funciones](#)
  - [8.3 Apuntadores y arreglos](#)
  - [8.4 Arreglos de apuntadores](#)
  - [8.5 Arreglos multidimensionales y apuntadores](#)
  - [8.6 Inicialización estática de arreglos de apuntadores](#)
  - [8.7 Apuntadores y estructuras](#)
  - [8.8 Fallas comunes con apuntadores](#)
  - [8.9 Ejercicios](#)
- 

# 8. Apuntadores

Los apuntadores son una parte fundamental de C. Si usted no puede usar los apuntadores apropiadamente entonces esta perdiendo la potencia y la flexibilidad que C ofrece básicamente. El secreto para C esta en el uso de apuntadores.

C usa los apuntadores en forma extensiva. ¿Porqué?

- Es la única forma de expresar algunos cálculos.
- Se genera código compacto y eficiente.
- Es una herramienta muy poderosa.

C usa apuntadores explícitamente con:

- Es la única forma de expresar algunos cálculos.
- Se genera código compacto y eficiente.
- Es una herramienta muy poderosa.

C usa apuntadores explícitamente con:

- Arreglos,
- Estructuras y
- Funciones

## 8.1 Definición de un apuntador

Un apuntador es una *variable* que contiene la dirección en memoria de otra variable. Se pueden tener apuntadores a cualquier tipo de variable.

El operador *unario* o *monádico* & devuelve la dirección de memoria de una variable.

El operador de *indirección* o *dereferencia* \* devuelve el ``contenido de un objeto apuntado por un apuntador".

Para declarar un apuntador para una variable entera hacer:

```
int *apuntador;
```

Se debe asociar a cada apuntador un tipo particular. Por ejemplo, no se puede asignar la dirección de un `short int` a un `long int`.

Para tener una mejor idea, considerar el siguiente código:

```
main()
{
    int x = 1, y = 2;
    int *ap;

    ap = &x;

    y = *ap;

    x = ap;

    *ap = 3;
}
```

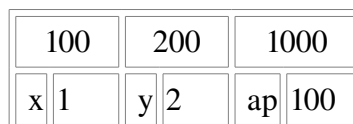
Cuando se compile el código se mostrará el siguiente mensaje:

```
warning: assignment makes integer from pointer without a cast.
```

Con el objetivo de entender el comportamiento del código supongamos que la variable `x` esta en la localidad de la memoria 100, `y` en 200 y `ap` en 1000. **Nota:** un apuntador es una variable, por lo tanto, sus valores necesitan ser guardados en algún lado.

```
int x = 1, y = 2;
int *ap;

ap = &x;
```



Las variables `x` e `y` son declaradas e inicializadas con 1 y 2 respectivamente, `ap` es declarado como un apuntador a entero y se le asigna la dirección de `x` (`&x`). Por lo que `ap` se carga con el valor 100.

```
y = *ap;
```



Después `y` obtiene el contenido de `ap`. En el ejemplo `ap` apunta a la localidad de memoria 100 -- la localidad de `x`. Por lo tanto, `y` obtiene el valor de `x` -- el cual es 1.

```
x = ap;
```

100	200	1000
x 100	y 1	ap 100

Como se ha visto C no es muy estricto en la asignación de valores de diferente tipo (apuntador a entero). Así que es perfectamente legal (aunque el compilador genera un aviso de cuidado) asigna el valor actual de `ap` a la variable `x`. El valor de `ap` en ese momento es 100.

```
*ap = 3;
```

100	200	1000
x 3	y 1	ap 100

Finalmente se asigna un valor al contenido de un apuntador (`*ap`).

**Importante:** Cuando un apuntador es declarado apunta a algún lado. Se debe inicializar el apuntador antes de usarlo. Por lo que:

```
main()
{
    int *ap;
    *ap = 100;
}
```

puede generar un error en tiempo de ejecución o presentar un comportamiento errático.

El uso correcto será:

```
main()
{
    int *ap;
    int x;

    ap = &x;
    *ap = 100;
}
```

Con los apuntadores se puede realizar también aritmética entera, por ejemplo:

```
main()
{
    float *flp, *flq;

    *flp = *flp + 10;

    ++*flp;

    (*flp)++;

    flq = flp;
}
```

**NOTA:** Un apuntador a cualquier tipo de variables es una dirección en memoria -- la cual es una dirección entera, pero un apuntador NO es un entero.

La razón por la cual se asocia un apuntador a un tipo de dato, es por que se debe conocer en cuantos bytes esta guardado el dato. De tal forma, que cuando se incrementa un apuntador, se incrementa el apuntador por un "bloque" de memoria, en donde el bloque esta en función del tamaño del dato.

Por lo tanto para un apuntador a un char, se agrega un byte a la dirección y para un apuntador a entero o a flotante se agregan 4 bytes. De esta forma si a un apuntador a flotante se le suman 2, el apuntador entonces se mueve dos posiciones float que equivalen a 8 bytes.

## 8.2 Apuntadores y Funciones

Cuando C pasa argumentos a funciones, los pasa *por valor*, es decir, si el parámetro es modificado dentro de la función, una vez que termina la función el valor pasado de la variable permanece inalterado.

Hay muchos casos que se quiere alterar el argumento pasado a la función y recibir el nuevo valor una vez que la función ha terminado. Para hacer lo anterior se debe usar una *llamada por referencia*, en C se puede simular pasando un puntero al argumento. Con esto se provoca que la computadora pase la dirección del argumento a la función.

Para entender mejor lo anterior consideremos la función `swap()` que intercambia el valor de dos argumentos enteros:

```
void swap(int *px, int *py);

main()
{
    int x, y;

    x = 10;
    y = 20;
    printf("x=%d\ty=%d\n", x, y);
    swap(&x, &y);
    printf("x=%d\ty=%d\n", x, y);
}

void swap(int *px, int *py)
{
    int temp;

    temp = *px;    /* guarda el valor de la direccion x */
    *px = *py;    /* pone y en x */
    *py = temp;   /* pone x en y */
}
```

## 8.3 Apuntadores y arreglos

Existe una relación estrecha entre los punteros y los arreglos. En C, un nombre de un arreglo es un índice a la dirección de comienzo del arreglo. En esencia, el nombre de un arreglo es un puntero al arreglo. Considerar lo siguiente:

```
int a[10], x;
int *ap;
```

```

ap = &a[0];    /* ap apunta a la direccion de a[0] */
x = *ap;      /* A x se le asigna el contenido de ap (a[0] en este caso) */
*(ap + 1) = 100; /* Se asigna al segundo elemento de 'a' el valor 100 usando
ap*/

```

Como se puede observar en el ejemplo la sentencia **a[t]** es idéntica a **ap+t**. Se debe tener cuidado ya que C no hace una revisión de los límites del arreglo, por lo que se puede ir fácilmente más allá del arreglo en memoria y sobrescribir otras cosas.

C sin embargo es mucho más sutil en su relación entre arreglos y apuntadores. Por ejemplo se puede teclear solamente:

```

ap = a; en vez de ap = &a[0]; y también *(a + i) en vez de a[i], esto es,
&a[i] es equivalente con a+i.

```

Y como se ve en el ejemplo, el direccionamiento de apuntadores se puede expresar como:

```

a[i] que es equivalente a *(ap + i)

```

Sin embargo los apuntadores y los arreglos son diferentes:

- Un apuntador es una variable. Se puede hacer `ap = a` y `ap++`.
- Un arreglo NO ES una variable. Hacer `a = ap` y `a++` ES ILEGAL.

Este parte es muy importante, asegúrese haberla entendido.

Con lo comentado se puede entender como los arreglos son pasados a las funciones. Cuando un arreglo es pasado a una función lo que en realidad se le esta pasando es la localidad de su elemento inicial en memoria.

Por lo tanto:

```

strlen(s) es equivalente a strlen(&s[0])

```

Esta es la razón por la cual se declara la función como:

```

int strlen(char s[]); y una declaración equivalente es int strlen(char
*s);

```

ya que `char s[]` es igual que `char *s`.

La función `strlen()` es una función de la *biblioteca estándar* que regresa la longitud de una cadena. Se muestra enseguida la versión de esta función que podría escribirse:

```

int strlen(char *s)
{
    char *p = s;

    while ( *p != '\0' )
        p++;
}

```

```

    return p - s;
}

```

Se muestra enseguida una función para copiar una cadena en otra. Al igual que en el ejercicio anterior existe en la biblioteca estándar una función que hace lo mismo.

```

void strcpy(char *s, char *t)
{
    while ( (*s++ = *t++) != '\0' );
}

```

En los dos últimos ejemplos se emplean apuntadores y asignación por valor. **Nota:** Se emplea el uso del caracter nulo con la sentencia `while` para encontrar el fin de la cadena.

## 8.4 Arreglos de apuntadores

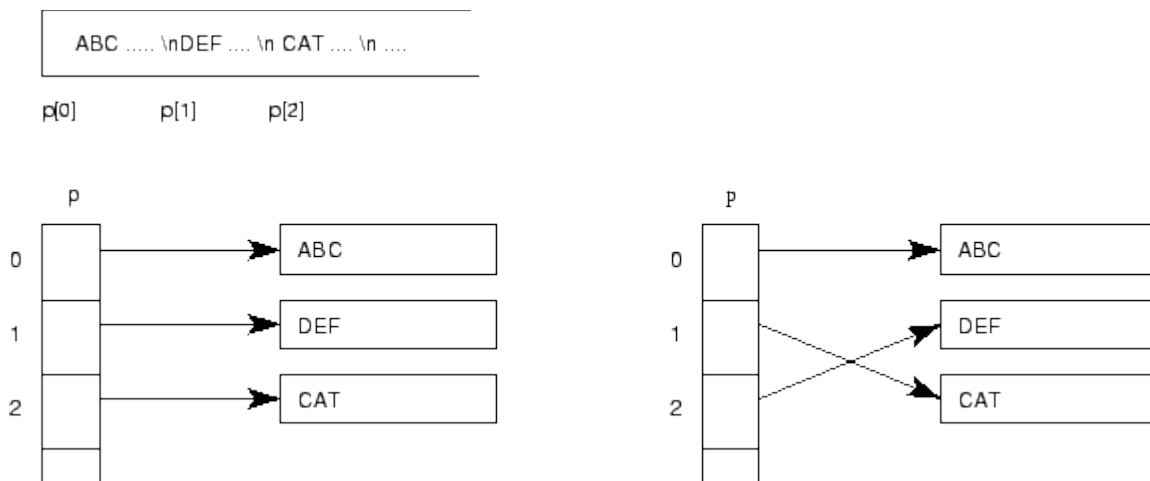
En C se pueden tener arreglos de apuntadores ya que los apuntadores son variables.

A continuación se muestra un ejemplo de su uso: **ordenar las líneas de un texto de diferente longitud.**

Los arreglos de apuntadores son una representación de datos que manejan de una forma eficiente y conveniente líneas de texto de longitud variable.

¿Cómo se puede hacer lo anterior?

1. Guardar todas las líneas en un arreglo de tipo `char` grande. Observando que `\n` marca el fin de cada línea. Ver figura [8.1](#).
2. Guardar los apuntadores en un arreglo diferente donde cada apuntador apunta al primer caracter de cada línea.
3. Comparar dos líneas usando la función de la biblioteca estándar `strcmp()`.
4. Si dos líneas están desacomodadas -- intercambiar (swap) los apuntadores (no el texto).



**Figura 8.1:** Arreglos de apuntadores (Ejemplo de ordenamiento de cadenas).

Con lo anterior se elimina:

- el manejo complicado del almacenamiento.
- alta sobrecarga por el movimiento de líneas.

## 8.5 Arreglos multidimensionales y apuntadores

Un arreglo multidimensional puede ser visto en varias formas en C, por ejemplo:

*Un arreglo de dos dimensiones es un arreglo de una dimensión, donde cada uno de los elementos es en sí mismo un arreglo.*

Por lo tanto, la notación

`a[n][m]`

nos indica que los elementos del arreglo están guardados renglón por renglón.

Cuando se pasa un arreglo bidimensional a una función se debe especificar el número de columnas -- el número de renglones es irrelevante.

La razón de lo anterior, es nuevamente los apuntadores. C requiere conocer cuantas son las columnas para que pueda brincar de renglón en renglón en la memoria.

Considerando que una función deba recibir `int a[5][35]`, se puede declarar el argumento de la función como:

```
f( int a[][35] ) { ..... }
```

o aún

```
f( int (*a)[35] ) { ..... }
```

En el último ejemplo se requieren los paréntesis `(*a)` ya que `[]` tiene una precedencia más alta que `*`.

Por lo tanto:

`int (*a)[35];` declara un apuntador a un arreglo de 35 enteros, y por ejemplo si hacemos la siguiente referencia `a+2`, nos estaremos refiriendo a la dirección del primer elemento que se encuentran en el tercer renglón de la matriz supuesta, mientras que

`int *a[35];` declara un arreglo de 35 apuntadores a enteros.

Ahora veamos la diferencia (sutil) entre apuntadores y arreglos. El manejo de cadenas es una aplicación común de esto.

Considera:

```
char *nomb[10];  
char anomb[10][20];
```

En donde es válido hacer `nomb[3][4]` y `anomb[3][4]` en C.

Sin embargo:

- `anomb` es un arreglo *verdadero* de 200 elementos de dos dimensiones tipo `char`.
- El acceso de los elementos `anomb` en memoria se hace bajo la siguiente fórmula  $20 * \text{ renglon} + \text{columna} + \text{dirección\_base}$
- En cambio `nomb` tiene 10 apuntadores a elementos.

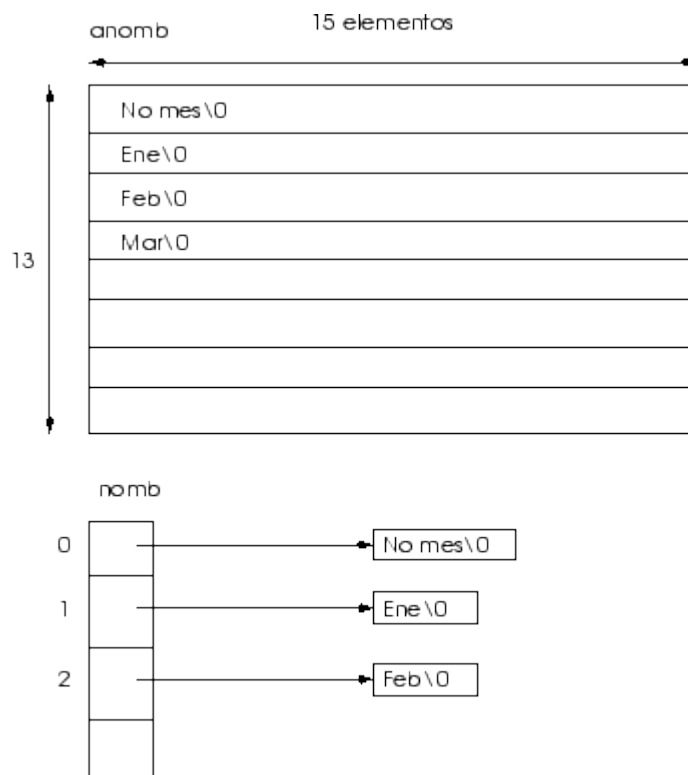
**NOTA:** si cada apuntador en `nomb` indica un arreglo de 20 elementos entonces y solamente entonces 200 chars estarán disponibles (10 elementos).

Con el primer tipo de declaración se tiene la ventaja de que cada apuntador puede apuntar a arreglos de diferente longitud.

Considerar:

```
char *nomb[] = { "No mes", "Ene", "Feb", "Mar", .... };  
char anomb[][15] = { "No mes", "Ene", "Feb", "Mar", ... };
```

Lo cual gráficamente se muestra en la figura 8.2. Se puede indicar que se hace un manejo más eficiente del espacio haciendo uso de un arreglo de apuntadores y usando un arreglo bidimensional.



**Figura 8.2:** Arreglo de 2 dimensiones VS. arreglo de apuntadores.

## 8.6 Inicialización estática de arreglos de apuntadores

La inicialización de arreglos de apuntadores es una aplicación ideal para un arreglo estático interno, por ejemplo:

```
func_cualquiera()
{
    static char *nomb[] = { "No mes", "Ene", "Feb", "Mar", .... };
}
```

Recordando que con el especificador de almacenamiento de clase *static* se reserva en forma permanente memoria el arreglo, mientras el código se esta ejecutando.

## 8.7 Apuntadores y estructuras

Los apuntadores a estructuras se definen fácilmente y en una forma directa. Considerar lo siguiente:

```
main()
{
    struct COORD { float x,y,z; } punto;

    struct COORD *ap_punto;

    punto.x = punto.y = punto.z = 1;

    ap_punto = &punto; /* Se asigna punto al apuntador */

    ap_punto->x++;      /* Con el operador -> se accesan los miembros */
    ap_punto->y+=2;      /* de la estructura apuntados por ap_punto */
    ap_punto->z=3;
}
```

Otro ejemplo son las listas ligadas:

```
typedef struct {
    int valor;
    struct ELEMENTO *sig;
} ELEMENTO;

ELEMENTO n1, n2;

n1.sig = &n2;
```

La asignación que se hace corresponde a la figura [8.3](#)



**Figura 8.3:** Esquema de una lista ligada con 2 elementos.

**Nota:** Solamente se puede declarar `sig` como un apuntador tipo `ELEMENTO`. No se puede tener un elemento del tipo variable ya que esto generaría una definición recursiva la cual no esta permitida. Se permite poner una referencia a un apuntador ya que los bytes se dejan de lado para cualquier apuntador.

## 8.8 Fallas comunes con apuntadores

A continuación se muestran dos errores comunes que se hacen con los apuntadores.

- **No asignar un apuntador a una dirección de memoria antes de usarlo**

```
int *x
*x = 100;
```

lo adecuado será, tener primeramente una localidad física de memoria, digamos `int y`;

```
int *x, y;
x = &y;
*x = 100;
```

- **Indirección no válida**

Supongamos que se tiene una función llamada `malloc()` la cual trata de asignar memoria dinámicamente (en tiempo de ejecución), la cual regresa un apuntador al bloque de memoria requerida si se pudo o un apuntador a nulo en otro caso.

```
char *malloc() -- una función de la biblioteca estándar que se verá más adelante.
```

Supongamos que se tiene un apuntador `char *p`

Considerar:

```
*p = (char *) malloc(100); /* pide 100 bytes de la memoria */
*p = 'y';
```

Existe un error en el código anterior. ¿Cuál es?

El `*` en la primera línea ya que `malloc` regresa un apuntador y `*p` no apunta a ninguna dirección.

El código correcto deberá ser:

```
p = (char *) malloc(100);
```

Ahora si `malloc` no puede regresar un bloque de memoria, entonces `p` es nulo, y por lo tanto no se podrá hacer:

```
*p = 'y';
```

Un buen programa en C debe revisar lo anterior, por lo que el código anterior puede ser reescrito como:

```
p = (char *) malloc(100);    /* pide 100 bytes de la memoria */

if ( p == NULL )
{
    printf("Error: fuera de memoria\n");
    exit(1);
}

*p = 'y';
```

## 8.9 Ejercicios

1. Escribir el programa que ordena las líneas de un texto leído desde la entrada estándar, donde cada línea tiene diferente longitud, según lo descrito en la sección de [arreglo de apuntadores](#).
2. Escribir una función que convierta una cadena *s* a un número de punto flotante usando apuntadores. Considerar que el número tiene el siguiente formato 99999999.999999, es decir, no se dará en notación científica. La función deberá suministrársele una cadena y deberá devolver un número.
3. Escribir un programa que encuentre el número de veces que una palabra dada (esto es, una cadena corta) ocurre en una sentencia (una cadena larga).

Leer los datos de la entrada estándar. La primera línea es una sola palabra, en la segunda línea se tiene un texto general. Leer ambas hasta encontrar un caracter de nueva línea. Recordar que se debe insertar un caracter nulo antes de procesar.

La salida típica podría ser:

```
La palabra es "el"
La sentencia es "el perro, el gato y el canario"
La palabra ocurrio 3 veces.
```

---

---

## Subsecciones

- [9.1 Uso de malloc, sizeof y free](#)
  - [9.2 calloc y realloc](#)
  - [9.3 Listas ligadas](#)
  - [9.4 Programa de revisión](#)
  - [9.5 Ejercicios](#)
- 

# 9. Asignación dinámica de memoria y Estructuras dinámicas

La asignación dinámica de memoria es una característica de C. Le permite al usuario crear tipos de datos y estructuras de cualquier tamaño de acuerdo a las necesidades que se tengan en el programa.

Se revisarán dos de las aplicaciones más comunes:

- Arreglos dinámicos
- Estructuras dinámicas de datos.

## 9.1 Uso de malloc, sizeof y free

La función `malloc` es empleada comúnmente para intentar "tomar" una porción contigua de memoria. Esta definida como:

```
void *malloc(size_t size);
```

Lo anterior indica que regresará un apuntador del tipo `void *`, el cual es el inicio en memoria de la porción reservada de tamaño `size`. Si no puede reservar esa cantidad de memoria la función regresa un apuntador nulo o `NULL`

Dado que `void *` es regresado, C asume que el apuntador puede ser convertido a cualquier tipo. El tipo de argumento `size_t` esta definido en la cabecera `stddef.h` y es un tipo entero sin signo.

Por lo tanto:

```
char *cp;  
  
cp = (char *) malloc(100);
```

intenta obtener 100 bytes y asignarlos a la dirección de inicio a `cp`.

Es usual usar la función `sizeof()` para indicar el número de bytes, por ejemplo:

```
int *ip;

ip = (int *) malloc(100 * sizeof(int) );
```

El compilador de C requiere hacer una conversión del tipo. La forma de lograr la coerción (cast) es usando `(char *)` y `(int *)`, que permite convertir un apuntador `void` a un apuntador tipo `char` e `int` respectivamente. Hacer la conversión al tipo de apuntador correcto asegura que la aritmética con el apuntador funcionará de forma correcta.

Es una buena práctica usar `sizeof()` aún si se conoce el tamaño actual del dato que se requiere, -- ya que de esta forma el código se hace independiente del dispositivo (portabilidad).

La función `sizeof()` puede ser usada para encontrar el tamaño de cualquier tipo de dato, variable o estructura. Simplemente se debe proporcionar uno de los anteriores como argumento a la función.

Por lo tanto:

```
int i;
struct COORD {float x,y,z};
struct COORD *pt;

sizeof(int), sizeof(i), sizeof(struct COORD) y
sizeof(PT) son tambien sentencias correctas.
```

En el siguiente ejemplo se reserva memoria para la variable `ip`, en donde se emplea la relación que existe entre apuntadores y arreglos, para manejar la memoria reservada como un arreglo. Por ejemplo, se pueden hacer cosas como:

```
main()
{
    int *ip, i;

    ip = (int *) malloc(100 * sizeof(int) );

    ip[0] = 1000;

        for (i=0; i<100; ++i)
            scanf("%d", ip++);
}
```

Cuando se ha terminado de usar una porción de memoria siempre se deberá liberar usando la función `free()`. Esta función permite que la memoria liberada este disponible nuevamente quizás para otra llamada de la función `malloc()`

La función `free()` toma un apuntador como un argumento y libera la memoria a la cual el apuntador hace referencia.

## 9.2 calloc y realloc

Existen dos funciones adicionales para reservar memoria, `calloc()` y `realloc()`. Los prototipos son dados a continuación:

```
void *calloc(size_t nmemb, size_t size);  
void *realloc(void *ptr, size_t size);
```

Cuando se usa la función `malloc()` la memoria no es inicializada (a *cero*) o borrada. Si se quiere inicializar la memoria entonces se puede usar la función `calloc`. La función `calloc` es computacionalmente un poco más cara pero, ocasionalmente, más conveniente que `malloc`. Se debe observar también la diferencia de sintaxis entre `calloc` y `malloc`, ya que `calloc` toma el número de elementos deseados (`nmemb`) y el tamaño del elemento (`size`), como dos argumentos individuales.

Por lo tanto para asignar a 100 elementos enteros que estén inicializados a cero se puede hacer:

```
int *ip;  
ip = (int *) calloc(100, sizeof(int) );
```

La función `realloc` intenta cambiar el tamaño de un bloque de memoria previamente asignado. El nuevo tamaño puede ser más grande o más pequeño. Si el bloque se hace más grande, entonces el contenido anterior permanece sin cambios y la memoria es agregada al final del bloque. Si el tamaño se hace más pequeño entonces el contenido sobrante permanece sin cambios.

Si el tamaño del bloque original no puede ser redimensionado, entonces `realloc` intentará asignar un nuevo bloque de memoria y copiará el contenido anterior. Por lo tanto, la función devolverá un nuevo apuntador (o de valor diferente al anterior), este nuevo valor será el que deberá usarse. Si no puede ser reasignada nueva memoria la función `realloc` devuelve `NULL`.

Si para el ejemplo anterior, se quiere reasignar la memoria a 50 enteros en vez de 100 apuntados por `ip`, se hará;

```
ip = (int *) realloc ( ip, 50*sizeof(int) );
```

## 9.3 Listas ligadas

Regresando al ejemplo del capítulo anterior se definió la estructura:

```
typedef struct {  
    int valor;  
    struct ELEMENTO *sig;  
} ELEMENTO;
```

La cual puede crecer en forma dinámica.

```
ELEMENTO *liga;
```

```
liga = (ELEMENTO *) malloc( sizeof(ELEMENT) ); /* Asigna memoria para liga */
free(liga); /* libera la memoria asignada al apuntador liga usando free() */
```

## 9.4 Programa de revisión

La *cola* es una colección de ordenada de elementos de la que se pueden borrar elementos en un extremo (llamado el *frente* de la cola) o insertarlos en el otro (llamado el *final* de la cola).

Se muestra a continuación el código completo para manipular esta estructura:

```
/* cola.c */
/* Demo de estructuras dinamicas en C */

#include <stdio.h>

#define FALSO 0

typedef struct nodo {
    int dato;
    struct nodo *liga;
} elemento_lista;

void Menu (int *opcion);
elemento_lista * AgregaDato (elemento_lista * apuntlista, int dato);
elemento_lista * BorrarDato (elemento_lista * apuntlista);
void ImprCola (elemento_lista * apuntlista);
void LimpCola (elemento_lista * apuntlista);

main ()
{
    elemento_lista listmember, *apuntlista;
    int dato, opcion;

    apuntlista = NULL;
    do {
        Menu (&opcion);
        switch (opcion) {
            case 1:
                printf ("Ingresa un dato que sera agregado ");
                scanf ("%d", &dato);
                apuntlista = AgregaDato (apuntlista, dato);
                break;
            case 2:
                if (apuntlista == NULL)
                    printf ("Cola vacia!\n");
                else
                    apuntlista = BorrarDato (apuntlista);
                break;
            case 3:
                ImprCola (apuntlista);
                break;

            case 4:
                break;

            default:
                printf ("Opcion no valida - intentar nuevamente\n");
                break;
        }
    } while (opcion != 4);
}
```

```

    LimpCola (apuntlista);
}
/* fin de main */

void Menu (int *opcion)
{
    char    local;

    printf("\nEntre\t1 para agregar un dato,\n\t2 para borrar un dato,\n\t3 para
mostrar el contenido de la cola\n\t4 para salir\n");
    do {
        local = getchar ();
        if ((isdigit (local) == FALSO) && (local != '\n'))
        {
            printf ("\nSe debe ingresar un entero.\n");
            printf ("Teclee 1 para agregar, 2 para borrar, 3 para imprimir, 4
para salir\n");
        }
    } while (isdigit ((unsigned char) local) == FALSO);
    *opcion = (int) local - '0';
}

elemento_lista *AgregaDato (elemento_lista *apuntlista, int dato)
{
    elemento_lista * lp = apuntlista;

    if (apuntlista != NULL) {
        while (apuntlista -> liga != NULL)
            apuntlista = apuntlista -> liga;
        apuntlista -> liga = (struct nodo *) malloc (sizeof (elemento_lista));
        apuntlista = apuntlista -> liga;
        apuntlista -> liga = NULL;
        apuntlista -> dato = dato;
        return lp;
    }
    else
    {
        apuntlista = (struct nodo *) malloc (sizeof (elemento_lista));
        apuntlista -> liga = NULL;
        apuntlista -> dato = dato;
        return apuntlista;
    }
}

elemento_lista *BorrarDato (elemento_lista *apuntlista)
{
    elemento_lista *tempp;
    printf ("El elemento borrado es %d\n", apuntlista -> dato);
    tempp = apuntlista -> liga;
    free (apuntlista);
    return tempp;
}

void ImprCola (elemento_lista *apuntlista)
{
    if (apuntlista == NULL)
        printf ("La cola esta vacia !!\n");
    else
        while (apuntlista != NULL) {
            printf ("%d\t", apuntlista -> dato);
            apuntlista = apuntlista -> liga;
        }
    printf ("\n");
}

```

```

}

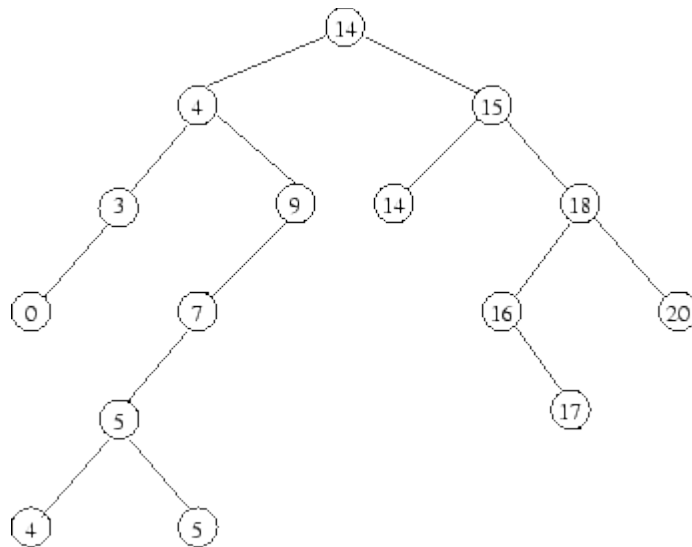
void LimpCola (elemento_lista *apuntlista)
{
    while (apuntlista != NULL) {
        apuntlista = BorrarDato (apuntlista);
    }
}

```

## 9.5 Ejercicios

1. Escribir un programa que lea un número, que indica cuántos números enteros serán guardados en un arreglo, crear el arreglo para almacenar el tamaño exacto de los datos y entonces leer los enteros que serán guardados en el arreglo.
2. Escribir un programa para ordenar una secuencia de números usando un árbol binario. Un árbol binario es una estructura tipo árbol con solamente 2 (posibles) ramas de cada nodo. Cada rama entonces representa una decisión de falso o verdadero. Para ordenar los números simplemente asignar a la rama izquierda los números menores respecto al número del nodo, y en la rama derecha el resto (es decir, los que son mayores o iguales a).

Por lo tanto, si la lista de entrada es: 14 15 4 9 7 18 3 5 16 4 20 17 0 14 5, se debe generar el árbol de la figura [9.1](#).



**Figura 9.1:** Arbol binario.

Para obtener una lista ordenada en forma ascendente, recorrer el árbol en preorden (*depth-first order*), es decir, visitar la raíz, recorrer el subárbol izquierdo en orden y recorrer el subárbol derecho en orden. Por lo que la salida deberá ser:

Los valores ordenados son:  
0 3 4 4 5 5 7 9 14 14 15 16 17 18 20

Mostrar 10 valores por línea.

---

---

## Subsecciones

- [10.1 Apuntadores a apuntadores](#)
  - [10.2 Entrada en la línea de comandos](#)
  - [10.3 Apuntadores a funciones](#)
  - [10.4 Ejercicios](#)
- 

# 10. Tópicos avanzados con apuntadores

Se han revisado varias aplicaciones y técnicas que usan apuntadores en los capítulos anteriores. Así mismo se han introducido algunos temas avanzados en el uso de apuntadores. En este capítulo se profundizan algunos tópicos que ya han sido mencionados brevemente y otros que completan la revisión de apuntadores en C.

En este capítulo se desarrolla lo siguiente:

- Se examinan apuntadores a apuntadores con más detalle.
- Como se usan los apuntadores en la línea de entrada.
- Y se revisan los apuntadores a funciones.

## 10.1 Apuntadores a apuntadores

Un arreglo de apuntadores es lo mismo que apuntadores a apuntadores. El concepto de arreglos de apuntadores es directo ya que el arreglo mantiene su significado claro. Sin embargo, se pueden confundir los apuntadores a apuntadores.

Un apuntador a un apuntador es una forma de *direccionamiento indirecto múltiple*, o una cadena de apuntadores. Como se ve en la figura [10.1](#), en el caso de un apuntador normal, el valor del apuntador es la dirección de la variable que contiene el valor deseado. En el caso de un apuntador a un apuntador, el primer apuntador contiene la dirección del segundo apuntador, que apunta a la variable que contiene el valor deseado.

Se puede llevar direccionamiento indirecto múltiple a cualquier extensión deseada, pero hay pocos casos donde más de un apuntador a un apuntador sea necesario, o incluso bueno de usar. La dirección indirecta en exceso es difícil de seguir y propensa a errores conceptuales.

Se puede tener un apuntador a otro apuntador de cualquier tipo. Considere el siguiente código:

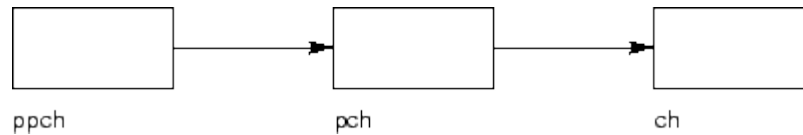
```
main()
{
    char ch;      /* Un caracter */
    char *pch;   /* Un apuntador a caracter */
    char **ppch; /* Un apuntador a un apuntador a caracter */
```

```

ch   = 'A';
pch  = &ch;
ppch = &pch;
printf("%c\n", **ppch); /* muestra el valor de ch */
}

```

Lo anterior se puede visualizar como se muestra en la figura [10.1](#), en donde se observa que `**ppch` se refiere a la dirección de memoria de `*pch`, la cual a su vez se refiere a la dirección de memoria de la variable `ch`. Pero ¿qué significa lo anterior en la práctica?



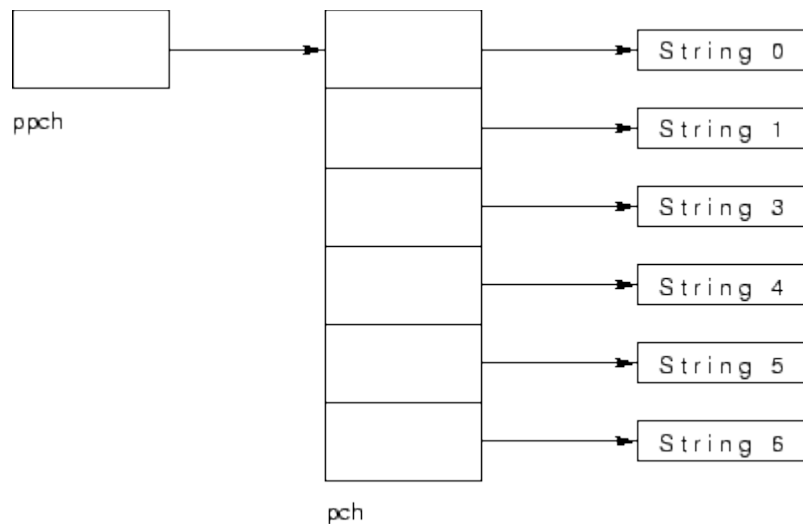
**Figura 10.1:** Apuntador a un apuntador, y apuntador a un char .

Se debe recordar que `char *` se refiere a una cadena la cual termina con un nulo. Por lo tanto, un uso común y conveniente es declarar un apuntador a un apuntador, y el apuntador a una cadena, ver figura [10.2](#).



**Figura 10.2:** Apuntador a un apuntador, y apuntador a una cadena.

Tomando un paso más allá lo anterior, se pueden tener varias cadenas apuntadas por el apuntador, ver figura [10.3](#)



**Figura 10.3:** Apuntador a varias cadenas.

Se pueden hacer referencias a cadenas individuales mediante `ppch[0]`, `ppch[1]`, .... Esto es idéntico a haber declarado `char *ppch[]`.

Una aplicación común de lo anterior es en los argumentos de la línea de comandos que se revisarán a continuación.

## 10.2 Entrada en la línea de comandos

C permite leer argumentos en la línea de comandos, los cuales pueden ser usados en los programas.

Los argumentos son dados o tecleados después del nombre del programa al momento de ser ejecutado el programa.

Lo anterior se ha visto al momento de compilar, por ejemplo:

```
gcc -o prog prog.c
```

donde `gcc` es el compilador y `-o prog prog.c` son los argumentos.

Para poder usar los argumentos en el código se debe definir como sigue la función `main`.

```
main(int argc, char **argv)
```

o

```
main(int argc, char *argv[])
```

Con lo que la función principal tiene ahora sus propios argumentos. Estos son solamente los únicos argumentos que la función `main` acepta.

\*

`argc` es el número de argumentos dados -- incluyendo el nombre del programa.

\*

`argv` es un arreglo de cadenas que tiene a cada uno de los argumentos de la línea de comandos -- incluyendo el nombre del programa en el primer elemento del arreglo.

Se muestra a continuación un programa de ejemplo:

```
main (int argc, char **argv)
{
    /* Este programa muestra los argumentos de la linea de comandos */
    int i;

    printf("argc = %d\n\n",argc);
    for (i=0; i<argc; ++i)
        printf("\t\targv[%d]: %s\n", i, argv[i]);
}
```

Suponiendo que se compila y se ejecuta con los siguientes argumentos:

```
args f1 "f2 y f3" f4 5 FIN
```

La salida será:

```
argc = 6
```

```
argv[0]: args
argv[1]: f1
argv[2]: f2 y f3
argv[3]: f4
```

```
argv[4]: 5
argv[5]: FIN
```

Observar lo siguiente:

- argv[0] contiene el nombre del programa.
- argc cuenta el número de argumentos incluyendo el nombre del programa.
- Los espacios en blanco delimitan el fin de los argumentos.
- Las comillas dobles " " son ignoradas y son usadas para incluir espacios dentro de un argumento.

## 10.3 Apuntadores a funciones

Los apuntadores a funciones son quizá uno de los usos más confusos de los apuntadores en C. Los apuntadores a funciones no son tan comunes como otros usos que tienen los apuntadores. Sin embargo, un uso común es cuando se pasan apuntadores a funciones como parámetros en la llamada a una función.

Lo anterior es especialmente útil cuando se deben usar distintas funciones quizás para realizar tareas similares con los datos. Por ejemplo, se pueden pasar los datos y la función que será usada por alguna función de *control*. Como se verá más adelante la biblioteca estándar de C da funciones para ordenamiento (*qsort*) y para realizar búsqueda (*bsearch*), a las cuales se les pueden pasar funciones.

Para declarar un apuntador a una función se debe hacer:

```
int (*pf) ();
```

Lo cual declara un apuntador `pf` a una función que regresa un tipo de dato `int`. Todavía no se ha indicado a que función apunta.

Suponiendo que se tiene una función `int f()`, entonces simplemente se debe escribir:

```
pf = &f;
```

para que `pf` apunte a la función `f()`.

Para que trabaje en forma completa el compilador es conveniente que se tengan los prototipos completos de las funciones y los apuntadores a las funciones, por ejemplo:

```
int f(int);
int (*pf) (int) = &f;
```

Ahora `f()` regresa un entero y toma un entero como parámetro.

Se pueden hacer cosas como:

```
ans = f(5);
ans = pf(5);
```

los cuales son equivalentes.

La función de la biblioteca estándar `qsort` es muy útil y esta diseñada para ordenar un arreglo usando un valor como *llave* de cualquier tipo para ordenar en forma ascendente.

El prototipo de la función `qsort` de la biblioteca `stdlib.h` es:

```
void qsort(void *base, size_t nmiemb, size_t tam,
           int (*compar)(const void *, const void *));
```

El argumento `base` apunta al comienzo del vector que será ordenado, `nmiemb` indica el tamaño del arreglo, `tam` es el tamaño en bytes de cada elemento del arreglo y el argumento final `compar` es un apuntador a una función.

La función `qsort` llama a la función `compar` la cual es definida por el usuario para comparar los datos cuando se ordenen. Observar que `qsort` conserva su independencia respecto al tipo de dato al dejarle la responsabilidad al usuario. La función `compar` debe regresar un determinado valor entero de acuerdo al resultado de comparación, que debe ser:

**menor que cero** : si el primer valor es menor que el segundo.

**cero** : si el primer valor es igual que el segundo.

**mayor que cero** : si el primer valor es mayor que el segundo.

A continuación se muestra un ejemplo que ordena un arreglo de caracteres, observar que en la función `comp`, se hace un *cast* para forzar el tipo `void *` al tipo `char *`.

```
#include <stdlib.h>

int comp(const void *i, const void *j);

main()
{
    int i;
    char cad[] = "facultad de ciencias fisico-matematicas";

    printf("\n\nArreglo original: \n");
    for (i=0; i<strlen(cad); i++)
        printf("%c", cad[i]);

    qsort(cad, strlen(cad), sizeof(char), comp );

    printf("\n\nArreglo ordenado: \n");
    for (i=0; i<strlen(cad); i++)
        printf("%c", cad[i]);

    printf("\n");
}

int comp(const void *i, const void *j)
{
    char *a, *b;
```

```
    a = (char *) i; /* Para forzar void * al tipo char *, se hace cast */
    b = (char *) j; /*      empleando (char *)      */
    return *a - *b;
}
```

## 10.4 Ejercicios

1. Escribir un programa que muestre las últimas líneas de un texto de entrada. Por defecto o ``default"  $n$  deberá ser 7, pero el programa deberá permitir un argumento opcional tal que  
`ultlin n`  
muestra las últimas  $n$  líneas, donde  $n$  es un entero. El programa deberá hacer el mejor uso del espacio de almacenamiento. (El texto de entrada podrá ser leído de un archivo dado desde la línea de comandos o leyendo un archivo de la entrada estándar.)
2. Escribir un programa que ordene una lista de enteros en forma ascendente. Sin embargo, si una opción  $r$  esta presente en la línea de comandos el programa deberá ordenar la lista en forma descendente.
3. Escribir un programa que lea la siguiente estructura y ordene los datos por la llave usando `qsort`

```
typedef struct {
    char llave[10];
    int algo_mas;
} Record;
```

---

---

## Subsecciones

- [11.1 Operadores sobre bits](#)
  - [11.2 Campos de bit](#)
    - [11.2.1 Portabilidad](#)
  - [11.3 Ejercicios](#)
- 

# 11. Operadores de bajo nivel y campos de bit

Se ha visto como los apuntadores nos dan control sobre las operaciones de bajo nivel de la memoria.

Muchos programas (por ejemplo, aplicaciones del tipo sistemas) operan actualmente a bajo nivel donde bits individuales deben ser manipulados.

La combinación de apuntadores y operadores a nivel bit hacen de C útil para muchas aplicaciones de bajo nivel y pueden casi reemplazar al código ensamblador. (Solamente un 10% aproximadamente de UNIX esta en código ensamblador el resto es C.)

## 11.1 Operadores sobre bits

Los operadores sobre bits de C se resumen en la siguiente tabla:

Operador	Acción
&	Y
	O
^	O exclusiva (XOR)
~	Complemento a uno
-	Negación
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha

No se debe confundir el operador & con el operador &&: & es el operador Y sobre bits, && es el operador lógico Y. Similarmente los operadores | y ||.

El operador unario ~ sólo requiere un argumento a la derecha del operador.

Los operadores de desplazamiento, >> y <<, mueven todos los bits en una posición hacia la derecha o la izquierda un determinado número de posiciones. El formato general de la sentencia de desplazamiento a la derecha es:

```
variable >> num_pos_de_bit
```

y el formato general de desplazamiento a la izquierda es

```
variable << num_pos_de_bit
```

Como los operadores desplazan bits en un sentido, la computadora trae ceros en el otro extremo. Se debe recordar que un desplazamiento *no* es una rotación: los bits desplazados en un extremo *no* vuelven al otro. Se pierden y los ceros traídos los reemplazan.

Una aplicación que tienen los operadores de desplazamiento de bits es para realizar multiplicaciones y divisiones rápidas con enteros. Como se ve en la siguiente tabla, donde un desplazamiento a la izquierda es multiplicar por 2 y uno a la derecha dividir por 2.

char x	Ejecución	Valor de x
x = 7;	0 0 0 0 0 1 1 1	7
x << 1;	0 0 0 0 1 1 1 0	14
x << 3;	0 1 1 1 0 0 0 0	112
x << 2;	1 1 0 0 0 0 0 0	192
x >> 1;	0 1 1 0 0 0 0 0	96
x >> 2;	0 0 0 1 1 0 0 0	24

Los desplazamientos son mucho más rápidos que la multiplicación actual (\*) o la división (/) por dos. Por lo tanto, si se quieren multiplicaciones o divisiones rápidas por 2 use desplazamientos.

Con la finalidad de ilustrar algunos puntos de los operadores sobre bits, se muestra la siguiente función `contbit`, la cual cuenta los bits puestos a 1 en un número de 8 bits (unsigned char) pasado como un argumento a la función.

```
int contbit(unsigned char x)
{
    int count;
    for (count=0; x!=0; x>>=1)
        if ( x & 1)
            count++;
    return count;
}
```

En esta función se ilustran varias características de C:

- El ciclo `for` no es usado para simplemente contar.
- `x>>=1` es equivalente a `x = x >> 1`.
- El ciclo `for` iterativamente desplaza a la derecha `x` hasta que `x` se hace 0.
- `x & 01` *enmascara* el primer bit de `x`, y si este es 1 entonces incrementa `count`.

## 11.2 Campos de bit

Al contrario de la mayoría de los lenguajes de programación, C tiene un método predefinido para acceder a un único bit en un byte. Este método puede ser utilísimo por una serie de razones:

- Si el almacenamiento es limitado, se pueden almacenar varias variables *booleanas* en un byte;
- ciertas interfaces de dispositivo transmiten información que se codifica en bits dentro de un byte;
- ciertas rutinas de encriptación necesitan acceder a los bits en un byte.

El método que C usa para acceder a los bits se basa en la estructura. Un campo de bit es un tipo especial de estructura que define la longitud en bits que tendrá cada elemento. El formato general de una definición de campo de bit es:

```
struct nomb_struct {  
  
    tipo nombre_1 : longitud;  
    tipo nombre_2 : longitud;  
    .  
    .  
    .  
    tipo nombre_n : longitud;  
  
}
```

Se debe declarar un campo de bit como **int**, **unsigned** o **signed**. Se debe declarar los campos de bits de longitud 1 como **unsigned**, ya que un bit único no puede tener signo.

Por ejemplo, considerar esta definición de estructura:

```
struct empaquetado {  
    unsigned int b1:1;  
    unsigned int b2:1;  
    unsigned int b3:1;  
    unsigned int b4:1;  
    unsigned int tipo:4;  
    unsigned int ent_raro:9;  
} paquete;
```

La estructura `empaquetado` contiene 6 miembros: 4 banderas de 1 bit (`b1`, `b2`, `b3` y `b4`), uno de 4 bits (`tipo`) y otro de 9 bits (`ent_raro`).

C automáticamente empaqueta los campos de bit anteriores tan compactamente como sea posible, donde la longitud máxima del campo es menor que o igual a la longitud de la palabra entera de la computadora. Si no fuera el caso, entonces algunos compiladores podrían permitir traslape en memoria para los campos, mientras otros podrían guardar el siguiente campo en la siguiente palabra.

La forma de acceder los miembros es en la forma usual:

```
paquete.tipo = 7;
```

Con estructura anterior se tiene que:

- Solamente los  $n$  bits más bajos son asignados a un número de  $n$  bits. Por lo tanto, el campo `tipo` no puede tomar valores mayores que 15, ya que es de 4 bits de largo.
- Los campos de bit son *siempre* convertidos a un tipo entero cuando se hace algún cálculo con ellos.
- Se permite mezclar los tipos "normales" con los campos de bit.

## 11.2.1 Portabilidad

Los campos de bit son una forma conveniente de expresar muchas operaciones difíciles. Sin embargo, los campos de bit carecen de portabilidad entre plataformas, por alguna de las siguientes razones:

- Los enteros podrían ser con o sin signo.
- Muchos compiladores limitan el número máximo de bits en el campo de bit al tamaño de un *integer*, el cual podría ser de 16 bits o de 32 bits.
- Algunos miembros campos de bit son guardados de izquierda a derecha, otros son guardados de derecha a izquierda en memoria.
- Si los campos de bits son muy largos, el siguiente campo de bit podría ser guardado consecutivamente en memoria (traslapando los límites entre las localidades de memoria) o en la siguiente palabra de memoria.

## 11.3 Ejercicios

1. Escribir una función que muestre un número de 8 bits (unsigned char) en formato binario.
2. Escribir una función `ponerbits(x, p, n, y)` que regrese `x`, con  $n$  bits que empiezan en la posición `p` y están a la derecha de una variable `y` unsigned char, dejándolos en `x` en la posición `p` y a la izquierda dejando los otros bits sin cambio.

Por ejemplo, si  $x = 10101010$  (170 decimal),  $y = 10100111$  (167 decimal),  $n = 3$  y  $p = 6$ , entonces se necesita tomar 3 bits de `y` (111) y ponerlos en `x` en la posición `10xxx010` para obtener la respuesta `10111010`.

La respuesta deberá mostrarse en forma binaria (ver primer ejercicio), sin embargo la entrada puede ser en forma decimal.

La salida podría ser como la siguiente:

```
x = 10101010 (binario)
y = 10100111 (binario)
ponerbits n = 3, p = 6 da x = 10111010 (binario)
```

3. Escribir una función que invierta los bits de  $x$  (unsigned char) y guarde la respuesta en  $y$ .

La respuesta deberá mostrarse en forma binaria (ver primer ejercicio), sin embargo la entrada puede estar en forma decimal.

La salida podría ser como la siguiente:

```
x = 10101010 (binario)
x invertida = 01010101 (binario)
```

4. Escribir una función que rote (no desplace) a la derecha  $n$  posiciones de bits de  $x$  del tipo unsigned char. La respuesta deberá mostrarse en forma binaria (ver primer ejercicio) y la entrada puede ser en forma decimal.

La salida podría ser como la siguiente:

```
x = 10100111 (binario)
x rotada por 3 = 11110100 (binario)
```

---

---

## Subsecciones

- [12.1 Directivas del preprocesador](#)
    - [12.1.1 #define](#)
    - [12.1.2 #undef](#)
    - [12.1.3 #include](#)
    - [12.1.4 #if Inclusión condicional](#)
  - [12.2 Control del preprocesador del compilador](#)
  - [12.3 Otras directivas del preprocesador](#)
  - [12.4 Ejercicios](#)
- 

# 12. El preprocesador de C

## 12.1 Directivas del preprocesador

Como se comento en el primer capítulo el preprocesamiento es el primer paso en la etapa de compilación de un programa -esta propiedad es única del compilador de C.

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

Las ventajas que tiene usar el preprocesador son:

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

### 12.1.1 #define

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

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

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

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

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

Por ejemplo:

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

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

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

main()
{
    int x=10, y=20;

    printf("EL minimo es %d\n", MIN(x,y) );
}
```

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

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

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

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

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

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

Otros ejemplos usando `#define` pueden ser:

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

#define IZQ_DESP_8 <<8
```

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

El uso de la sustitución de macros en el lugar de las funciones reales tiene un beneficio importante:

incrementa la velocidad del código porque no se penaliza con una llamada de función. Sin embargo, se paga este incremento de velocidad con un incremento en el tamaño del programa porque se duplica el código.

## 12.1.2 #undef

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

```
#undef <nombre de macro>
```

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

## 12.1.3 #include

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

```
#include <archivo>
```

```
#include "archivo"
```

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

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

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

## 12.1.4 #if Inclusión condicional

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

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

Por ejemplo,

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

```
#define PAIS_ACTIVO MEX

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

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

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

```
#ifdef <nombre de macro>
<secuencia de sentencias>
#endif
```

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

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

```
#ifndef <nombre de macro>
<secuencia de sentencias>
#endif
```

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

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

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

## 12.2 Control del preprocesador del compilador

Se puede usar el compilador `gcc` para controlar los valores dados o definidos en la línea de comandos. Esto permite alguna flexibilidad para configurar valores además de tener algunas otras funciones útiles. Para lo anterior, se usa la opción `-Dmacro [=defn]`, por ejemplo:

```
gcc -DLONGLINEA=80 prog.c -o prog
```

que hubiese tenido el mismo resultado que

```
#define LONGLINEA 80
```

en caso de que hubiera dentro del programa (`prog.c`) algún `#define` o `#undef` pasaría por encima de la entrada de la línea de comandos, y el compilador podría mandar un ```warning``` sobre esta situación.

También se puede poner un símbolo sin valor, por ejemplo:

```
gcc -DDEBUG prog.c -o prog
```

En donde el valor que se toma es de 1 para esta macro.

Las aplicaciones pueden ser diversas, como por ejemplo cuando se quiere como una bandera para depuración se puede hacer algo como lo siguiente:

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

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

```
x = y * 3;

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

La opción `-E` hace que la compilación se detenga después de la etapa de preprocesamiento, por lo anterior no se esta propiamente compilando el programa. La salida del preprocesamiento es enviada a la entrada estándar. GCC ignora los archivos de entrada que no requieran preprocesamiento.

## 12.3 Otras directivas del preprocesador

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

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

La directiva **#line número ``cadena``** informa al preprocesador cual es el número siguiente de la línea de entrada. Cadena es opcional y nombra la siguiente línea de entrada. Esto es usado frecuentemente cuando son traducidos otros lenguajes a C. Por ejemplo, los mensajes de error producidos por el compilador de C pueden referenciar el nombre del archivo y la línea de la fuente

original en vez de los archivos intermedios de C.

Por ejemplo, lo siguiente especifica que el contador de línea empezará con 100.

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

## 12.4 Ejercicios

1. Definir una macro `min(a, b)` para determinar el entero más pequeño. Definir otra macro `min3(a, b, c)` en términos de `min(a, b)`. Incorporar las macros en un programa demostrativo en donde se pida al usuario tres números y se muestre el más pequeño.
  2. Definir un nombre de macro de preprocesador para seleccionar:
    - los bits menos significativos de un tipo de dato `unsigned char`.
    - el  $n$ -ésimo bit (asumiendo que el menos significativo es 0) de un tipo de dato `unsigned char`.
- 
-

---

## Subsecciones

- [13.1 Ventajas del usar UNIX con C](#)
  - [13.2 Uso de funciones de bibliotecas y llamadas del sistema](#)
- 

# 13. C, UNIX y las bibliotecas estándar

Existe una relación estrecha entre C y el sistema operativo que ejecuta los programas de C. El sistema operativo UNIX esta escrito en su mayor parte con C. En este capítulo se verá como C y UNIX interactúan juntos.

Se usa UNIX para manejar el espacio del archivo, para editar, compilar o ejecutar programas, entre otras cosas.

Sin embargo UNIX es mucho más útil que lo anterior.

## 13.1 Ventajas del usar UNIX con C

- *Portabilidad* Unix o alguna variante de UNIX están disponibles en muchas máquinas. Los programas escritos con UNIX y C estándares deben correr en cualquier máquina prácticamente sin ningún problema.
- *Multiusuario/Multitarea* Muchos programas pueden compartir la capacidad de procesamiento de las máquinas.
- *Manejo de archivos* El sistema jerárquico de archivos emplea muchas rutinas de manejo de archivos.
- *Programación del Shell* UNIX suministra un intérprete de comandos poderoso que entiende mas de 200 comandos y que puede también correr UNIX o programas del usuario.
- *Entubamiento o Pipe* Permite la conexión entre programas, en donde la salida de un programa puede ser la entrada de otro. Lo anterior puede hacerse desde la línea de comandos o dentro de un programa de C.
- *Utilerías de UNIX* Hay cerca de 200 utilerías que permiten ejecutar muchas rutinas sin escribir nuevos programas. Por ejemplo: make, grep diff, awk, etc.
- *Llamadas al sistema* UNIX tiene aproximadamente 60 llamadas al sistema, que son el corazón del sistema operativo o del *kernel* de UNIX. Las llamadas están actualmente escritas en C. Todas ellas pueden ser accesadas desde programas de C. Ejemplos de estas son el sistema básico de E/S, acceso al reloj del sistema. La función `open()` es un ejemplo de una llamada al sistema.
- *Biblioteca de funciones* Que son adiciones al sistema operativo.

## 13.2 Uso de funciones de bibliotecas y llamadas del sistema

Para usar las bibliotecas de funciones y las llamadas al sistema en un programa de C simplemente se debe llamar la función apropiada de C.

Ejemplos de las funciones de la biblioteca estándar que han sido vistas son las funciones de E/S de alto nivel `-fprintf()`, `malloc()`, ...

Operadores aritméticos, generadores de números aleatorios -- `random()`, `srandom()`, `lrand48()`, `drand48()`, etc. y funciones para conversión de cadenas a los tipos básicos de C (`atoi()`, `atof()`, etc.) son miembros de la biblioteca estándar `stdlib.h`.

Todas las funciones matemáticas como `sin()`, `cos()`, `sqrt()` son funciones de la biblioteca estándar de matemáticas (`math.h`).

Para muchas llamadas del sistema y funciones de las bibliotecas se tiene que incluir el archivo cabecera apropiado, por ejemplo: `stdio.h`, `math.h`.

Para usar una función se debe asegurar de haber puesto los correspondientes `#include` en el archivo de C. De esta forma la función puede ser llamada correctamente.

Es importante asegurarse que los argumentos tengan los tipos esperados, de otra forma la función probablemente producirá resultados extraños.

Algunas bibliotecas requieren opciones extras antes de que el compilador pueda soportar su uso. Por ejemplo, para compilar un programa que incluya funciones de la biblioteca `math.h` el comando podría ser de la siguiente forma:

```
gcc matprog.c -o matprog.c -lm
```

La opción final `-lm` es una instrucción para ligar la biblioteca matemática con el programa. La página de `man` para cada función usualmente informa si se requiere alguna bandera de compilación especial.

Información de casi todas las llamadas al sistema y funciones de biblioteca están disponibles en las páginas del `man`. Se encuentran disponibles en línea con tan sólo teclear `man` y el nombre de la función. Por ejemplo:

```
man drand48
```

nos dará información acerca de éste generador de números aleatorios.

---

---

---

## Subsecciones

- [14.1 Funciones aritméticas](#)
  - [14.2 Números aleatorios](#)
  - [14.3 Conversión de cadenas](#)
  - [14.4 Búsqueda y ordenamiento](#)
  - [14.5 Ejercicios](#)
- 

# 14. Biblioteca <stdlib.h>

Para usar todas las funciones de ésta biblioteca se debe tener la siguiente directiva

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

Las funciones de la biblioteca pueden ser agrupadas en tres categorías básicas:

- Aritméticas;
- Números aleatorios; y
- Conversión de cadenas.

El uso de todas las funciones es sencillo. Se consideran dentro del capítulo en forma breve.

## 14.1 Funciones aritméticas

Hay cuatro funciones enteras básicas:

```
int abs(int j);
long int labs(long int j);
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
```

Fundamentalmente hay dos funciones para enteros y para compatibilidad con enteros largos.

- *abs()* La función regresa el valor absoluto del argumento entero *j*.
- *div()* Calcula el valor *numer* entre *denom* y devuelve el cociente y el resto en una estructura llamada *div\_t* que contiene dos miembros llamados *quot* y *rem*.

La estructura *div\_t* esta definida en *stdlib.h* como sigue:

```
typedef struct {
    int quot; /* cociente */
    int rem; /* residuo */
} div_t;
```

La estructura *ldiv\_t* es definida de una forma similar.

Se muestra un ejemplo donde se hace uso de la función `div_t`:

```
#include <stdlib.h>

main()
{
    int num=8, den=3;
    div_t res;

    res = div(num,den);

    printf("Respuesta:\n\t Cociente = %d\n\t Residuo = %d\n",
        res.quot, res.rem);
}
```

que genera la siguiente salida:

```
Respuesta:
    Cociente = 2
    Residuo = 2
```

## 14.2 Números aleatorios

Los números aleatorios son útiles en programas que necesitan simular eventos aleatorios, tales como juegos, simulaciones y experimentos. En la práctica ninguna función produce datos aleatorios verdaderos -- las funciones producen números **pseudo-aleatorios**. Los números aleatorios son calculados a partir de una fórmula dada (los distintos generadores usan diferentes fórmulas) y las secuencias de números que son producidas se repiten. Una **semilla (seed)** es usualmente inicializada para que la secuencia sea generada. Por lo tanto, si la semilla es siempre inicializada con el mismo valor todo el tiempo, el mismo conjunto será siempre calculado.

Una técnica común para introducir más aleatoriedad en el generador de números aleatorios es usando el tiempo y la fecha para inicializar la semilla, ya que éste siempre estará cambiando.

Hay muchas funciones de números (pseudo) aleatorios en la biblioteca estándar. Todas ellas operan con la misma idea básica pero generan secuencias diferentes de números (basadas en funciones generadoras diferentes) sobre rangos diferentes de números.

El conjunto más simple de funciones es:

```
int rand(void);
void srand(unsigned int semilla);
```

- `rand()` La función devuelve un entero pseudo-aleatorio entre 0 y  $2^{15} - 1$  (RAND\_MAX).
- `srand()` Establece su argumento como la semilla de una nueva serie de enteros pseudo-aleatorios.

Un ejemplo sencillo del uso del tiempo de la fecha es inicializando la semilla a través de una llamada:

```
srand( (unsigned int) time( NULL ) );
```

El siguiente programa `tarjeta.c` muestra el uso de estas funciones para simular un paquete de tarjetas que esta siendo revueltas.

```
/*
** Se usan numeros aleatorios para revolver las "tarjetas" de la baraja.
** El segundo argumento de la funcion indica el numero de tarjetas.
** La primera vez que esta funcion es llamada, srand es
** llamada para inicializar el generador de numeros aleatorios.
*/
#include <stdlib.h>
#include <time.h>
#define VERDADERO 1
#define FALSO 0

void intercambiar( int *baraja, int n_cartas )
{
    int i;
    static int primera_vez = VERDADERO;

    /*
    ** Inicializar el generador de numeros con la fecha actual
    ** si aun no se ha hecho.
    */
    if( primera_vez ){
        primera_vez = FALSO;
        srand( (unsigned int)time( NULL ) );
    }

    /*
    ** "intercambiar" empleando pares de cartas.
    */
    for( i = n_cartas - 1; i > 0; i -= 1 ){
        int alguno;
        int temp;

        alguno = rand() % i;
        temp = baraja[ alguno ];
        baraja[ alguno ] = baraja[ i ];
        baraja[ i ] = temp;
    }
}
```

## 14.3 Conversión de cadenas

Existen unas cuantas funciones para convertir cadenas a enteros, enteros largos y valores flotantes. Estas son:

- **double atof(const char \*cadena)** Convierte una cadena a un valor flotante.
- **int atoi(const char \*cadena)** Convierte una cadena a un valor entero.
- **int atol(const char \*cadena)** Convierte una cadena a un valor entero largo.
- **double strtod(const char \*cadena, char \*\*finap)** Convierte una cadena a un valor de punto flotante.
- **double strtol(const char \*cadena, char \*finap, int base)** Convierte una cadena a un entero largo de acuerdo a una base dada, la cual deberá estar entre 2 y 36 inclusive.
- **unsigned long strtoul(const char \*cadena, char \*finap, int base)** Convierte una cadena a un entero largo sin signo.

Varias de las funciones se pueden usar en forma directa, por ejemplo:

```
char *cad1 = "100";
char *cad2 = "55.444";
char *cad3 = "    1234";
char *cad4 = "123cuatro";
char *cad5 = "invalido123";
char *cad6 = "123E23Hola";
char *cad7;

int i;
float f;

i = atoi(cad1);    /* i = 100 */
f = atof(cad2);    /* f = 55.44 */
i = atoi(cad3);    /* i = 1234 */
i = atoi(cad4);    /* i = 123 */
i = atoi(cad5);    /* i = 0 */
f = strtod(cad6, &cad7); /* f=1.230000E+25 y cad7=hola*/
```

Nota:

- Los caracteres en blanco son saltados.
- Caracteres ilegales son ignorados.
- Si la conversión no puede ser hecha se regresa cero y `errno` es puesta con el valor `ERANGE`.

## 14.4 Búsqueda y ordenamiento

La biblioteca `stdlib.h` tiene dos funciones útiles para hacer búsqueda y ordenamiento de datos de cualquier tipo. La función `qsort()` ya fue vista previamente en la sección [10.3](#).

Para completar la lista se anota el prototipo, pero para ver un ejemplo pasar al capítulo indicado.

La función `qsort` de la biblioteca estándar es muy útil, ya que esta diseñada para ordenar un arreglo por un valor *llave* de *cualquier tipo* en orden ascendente, con tal de que los elementos del arreglo sean de un tipo fijo.

El prototipo de la función de acuerdo a `stdlib.h` es:

```
void qsort(void *base, size_t nmiemb, size_t tam,
           int (*compar)(const void *, const void *));
```

Similarmente, hay una función para búsqueda binaria, `bsearch()` la cual tiene el siguiente prototipo en `stdlib.h` como:

```
void *bsearch(const void *key, const void *base, size_t nmemb,
              size_t size, int (*compar)(const void *, const void *));
```

A continuación se muestra un ejemplo que hace uso de la función `bsearch()`:

```
#define TAM 10
#include <stdio.h>
#include <stdlib.h>
```

```

typedef struct {
    int llave;
    char informacion[30];
} Registro;

int compara_registro(void const *i, void const *j)
{
    int a, b;

    a = ((Registro *) i)->llave;
    b = ((Registro *) j)->llave;
    return a - b;
}

main()
{
    Registro llave;
    Registro *resp;
    Registro arreglo[TAM];
    int long_arreglo = TAM;

    arreglo[0].llave = -43; arreglo[1].llave = -9; arreglo[2].llave = 0;
    arreglo[3].llave = 3; arreglo[4].llave = 4; arreglo[5].llave = 9;
    arreglo[6].llave = 10; arreglo[7].llave = 12; arreglo[8].llave = 30;
    arreglo[9].llave = 203;

    printf("Introduce el valor de la llave que se buscara en el arreglo: ");
    scanf("%d",&llave.llave); /* valor del indice que sera buscado */
    resp = (Registro *) bsearch(&llave, arreglo, long_arreglo,
sizeof(Registro),
    compara_registro);
    if (resp != NULL )
        printf("Se localizo la llave %d\n",resp->llave);
    else
        printf("No se localizo ningun elemento con la llave %d\n",
llave.llave);
}

```

La función `bsearch()` regresa un apuntador al registro que coincide con la llave dada, o bien, el valor `NULL` si no encuentra el registro.

Observar que el tipo del argumento de la llave debe ser del mismo tipo que la del arreglo de los elementos (en nuestro caso `Registro`).

## 14.5 Ejercicios

1. Escribir un programa que simule el lanzamiento de un dado.
2. Escribir un programa que simule el melate, en donde se seleccionan 6 números enteros en un rango de 1 al 44
3. Escribir un programa que lea un número de la línea de entrada y genere un número flotante aleatorio en el rango de 0 a el número de entrada.

---

## Subsecciones

- [15.1 Funciones matemáticas](#)
  - [15.2 Constantes matemáticas](#)
- 

# 15. Biblioteca <math.h>

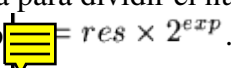
La biblioteca de matemáticas es relativamente fácil de usar, al igual que las vistas previamente. Se debe incluir la directiva de preprocesamiento `#include <math.h>`, además de recordar de ligar la biblioteca de matemáticas al compilar:

```
gcc progm1.c -o progm1 -lm
```

Un error común es el olvidar incluir el archivo `<math.h>`.

## 15.1 Funciones matemáticas

Se muestra a continuación una lista de funciones matemáticas. Son fáciles de usar y algunas de ellas han sido ya usadas previamente. No se proporciona ejemplo de las mismas.

- `double acos(double x)` Calcula el arco coseno de  $x$ .
- `double asin(double x)` Calcula el arco seno de  $x$ .
- `double atan(double x)` Devuelve el arco tangente en radianes.
- `double atan2(double y, double x)` Calcula el arco tangente de las dos variables  $x$  e  $y$ . Es similar a calcular el arco tangente de  $y/x$ , excepto en que los signos de ambos argumentos son usados para determinar el cuadrante del resultado.
- `double ceil(double x)` Redondea  $x$  hacia arriba al entero más cercano.
- `double cos(double x)` devuelve el coseno de  $x$ , donde  $x$  está dado en radianes.
- `double cosh(double x)` Devuelve el coseno hiperbólico de  $x$ .
- `double exp(double x)` Devuelve el valor de  $e$  (la base de los logaritmos naturales) elevado a la potencia  $x$ .
- `double fabs(double x)` Devuelve el valor absoluto del número en punto flotante  $x$ .
- `double floor(double x)` Redondea  $x$  hacia abajo al entero más cercano.
- `double fmod(double x, double y)` Calcula el resto de la división de  $x$  entre  $y$ . El valor devuelto es  $x - n * y$ , donde  $n$  es el cociente de  $x/y$ .
- `double frexp(double x, int *exp)` Se emplea para dividir el número  $x$  en una fracción normalizada y un exponente que se guarda en `exp`.  
$$x = res \times 2^{exp}$$
- `long int labs(long int j)` Calcula el valor absoluto de un entero largo.
- `double ldexp(double x, int exp)` Devuelve el resultado de multiplicar el número  $x$  por 2 elevado a  $exp$  (inversa de `frexp`).
- `double log(double x)`; Devuelve el logaritmo neperiano de  $x$ .
- `double log10(double x)` Devuelve el logaritmo decimal de  $x$ .

- `double modf(double x, double *iptr)` Divide el argumento `x` en una parte entera y una parte fraccional. La parte entera se guarda en `iptr`.
- `double pow(double x, double y)` Devuelve el valor de `x` elevado a `y`.
- `double sin(double x)` Devuelve el seno de `x`.
- `double sinh(double x)` Regresa el seno hiperbólico de `x`.
- `double sqrt(double x)` Devuelve la raíz cuadrada no negativa de `x`.
- `double tan(double x)` Devuelve la tangente de `x`.
- `double tanh(double x)` Devuelve la tangente hiperbólica de `x`.

## 15.2 Constantes matemáticas

La biblioteca de matemáticas define varias constantes (por lo general desechadas). Siempre es aconsejable usar estas definiciones.

- `M_E` La base de los logaritmos naturales  $e$ .
  - `M_LOG2E` El logaritmo de  $e$  de base 2.
  - `M_LOG10E` El logaritmo de  $e$  de base 10.
  - `M_LN2` El logaritmo natural de 2.
  - `M_LN10` El logaritmo natural de 10.
  - `M_PI`  $\pi$
  - `M_PI_2`  $\frac{\pi}{2}$
  - `M_PI_4`  $\frac{\pi}{4}$
  - `M_1_PI`  $\frac{1}{\pi}$
  - `M_2_PI`  $\frac{2}{\pi}$
  - `M_2_SQRTPI`  $\frac{2}{\sqrt{\pi}}$
  - `M_SQRT2` La raíz cuadrada positiva de 2
  - `M_SQRT1_2` La raíz cuadrada positiva de 1/2
-

---

## Subsecciones

- [16.1 Reportando errores](#)
  - [16.1.1 perror\(\)](#)
  - [16.1.2 errno](#)
  - [16.1.3 exit](#)
- [16.2 Flujos](#)
  - [16.2.1 Flujos predefinidos](#)
    - [16.2.1.1 Redireccionamiento](#)
- [16.3 E/S Basica](#)
- [16.4 E/S formateada](#)
  - [16.4.1 printf](#)
  - [16.4.2 scanf](#)
- [16.5 Archivos](#)
  - [16.5.1 Lectura y escritura de archivos](#)
- [16.6 sprintf y sscanf](#)
  - [16.6.1 Petición del estado del flujo](#)
- [16.7 E/S de bajo nivel o sin almacenamiento intermedio](#)
- [16.8 Ejercicios](#)

---

# 16. Entrada y salida (E/S) `stdio.h`

En este capítulo se verán varias formas de entrada y salida (E/S). Se han mencionado brevemente algunas formas y ahora se revisarán con un poco más de detalle.

Los programas que hagan uso de las funciones de la biblioteca de E/S deben incluir la *cabecera*, esto es:

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

## 16.1 Reportando errores

En muchas ocasiones es útil reportar los errores en un programa de C. La función de la biblioteca estándar `perror` es la indicada para hacerlo. Es usada conjuntamente con la variable `errno` y frecuentemente cuando se encuentra un error se desea terminar el programa antes. Además se revisa la función `exit()` y `errno`, que en un sentido estricto no son parte de la biblioteca `stdio.h`, para ver como trabajan con `perror`.

## 16.1.1 perror ( )

El prototipo de la función `perror` es:

```
void perror(const char *s);
```

La función `perror ( )` produce un mensaje que va a la salida estándar de errores, describiendo el último error encontrado durante una llamada al sistema o a ciertas funciones de biblioteca. La cadena de caracteres `s` que se pasa como argumento, se muestra primero, luego un signo de dos puntos y un espacio en blanco; por último, el mensaje y un salto de línea. Para ser de más utilidad, la cadena de caracteres pasada como argumento debería incluir el nombre de la función que incurrió en el error. El código del error se toma de la variable externa `errno`, que toma un valor cuando ocurre un error, pero no es puesta a cero en una llamada no errónea.

## 16.1.2 errno

A la variable especial del sistema `errno` algunas llamadas al sistema (y algunas funciones de biblioteca) le dan un valor entero, para indicar que ha habido un error. Esta variable está definida en la cabecera `#include <errno.h>` y para ser usada dentro de un programa debe ser declarada de la siguiente forma:

```
extern int errno;
```

El valor sólo es significativo cuando la llamada devolvió un error (usualmente -1), algunas veces una función también puede devolver -1 como valor válido, por lo que se debe poner `errno` a cero antes de la llamada, para poder detectar posibles errores.

## 16.1.3 exit

La función `exit` tiene el siguiente prototipo de acuerdo a la cabecera `#include <stdlib.h>`:

```
void exit(int status);
```

La función produce la terminación normal del programa y la devolución de `status` al proceso padre o al sistema operativo. El valor de `status` es usado para indicar como ha terminado el programa:

- o sale con un valor `EXIT_SUCCESS` en una terminación
- o sale con un valor `EXIT_FAILURE` en una terminación

Por lo tanto cuando se encuentre un error se llamará a la función como `exit (EXIT_FAILURE)` para terminar un programa con errores.

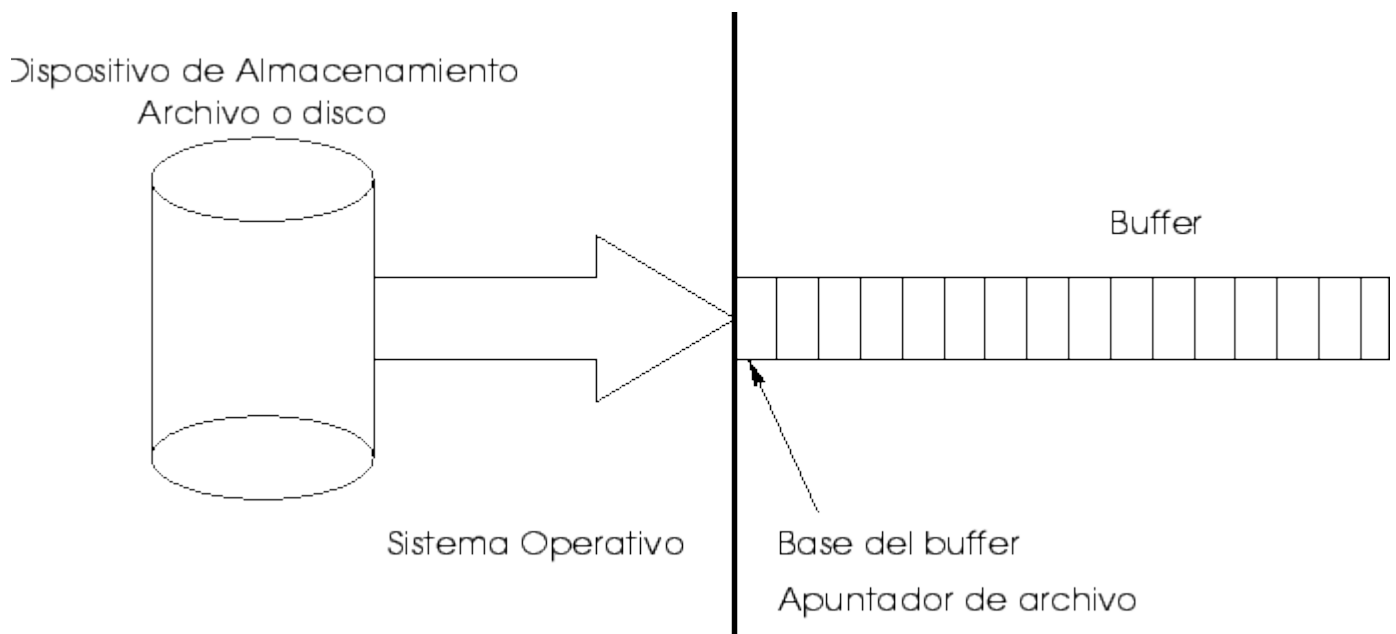
## 16.2 Flujos

Los **flujos** son una forma flexible y eficiente para leer y escribir datos.

Existe una estructura interna de C, `FILE`, la cual representa a todas los flujos y esta definida en `stdio.h`. Por lo tanto simplemente se necesita referirse a la estructura para realizar entrada y salida de datos.

Para usar los flujos entonces se debe declarar una variable o apuntador de este tipo en el programa. No se requiere conocer más detalles acerca de la definición. Se debe *abrir* un flujo antes de realizar cualquier E/S, después se puede acceder y entonces se cierra.

El flujo de E/S usa un *BUFFER*, es decir, un pedazo fijo de área temporal de la memoria (el buffer) es leído o escrito a un archivo. Lo siguiente se muestra en la figura [17.1](#). Observar que el apuntador del archivo actualmente apunta a éste buffer.



**Figura 16.1:** Modelo de entrada salida usando un buffer.

Esto conduce a un uso eficiente de E/S pero se debe tener cuidado: los datos escritos a un buffer no aparecen en un archivo (o dispositivo) hasta que el buffer es escrito (con `\n` se puede hacer). Cualquier salida anormal del código puede causar problemas.

### 16.2.1 Flujos predefinidos

En UNIX se tienen predefinidos 3 flujos (en `stdio.h`): `stdin`, `stdout` y `stderr`.

Todas ellas usan texto como método de E/S.

Los flujos `stdin` y `stdout` pueden ser usadas con archivos, programas, dispositivos de E/S como el teclado, la consola, etc. El flujo `stderr` siempre va a la consola o la pantalla.

La consola es el dispositivo predefinido para `stdout` y `stderr`. El teclado lo es para `stdin`.

Los flujos predefinidos son automáticamente abiertas.

### 16.2.1.1 Redireccionamiento

Lo siguiente no es parte de C, pero depende del sistema operativo. Se hace redireccionamiento desde la línea de comandos con:

> que redirecciona stdout a un archivo.

Por lo tanto, si se tiene un programa llamado *salida*, que usualmente muestra en pantalla algo, entonces:

```
salida > archivo_sal
```

mandará la salida al archivo *archivo\_sal*

< redirecciona stdin desde un archivo a un programa.

Por lo tanto, si se espera entrada desde el teclado para un programa llamado *entrada*, se puede leer en forma similar la entrada desde un archivo.

```
entrada < archivo_ent
```

Con | *entubamiento* o *pipe* se coloca stdout de un programa en stdin de otro,

```
prog1 | prog2
```

Por ejemplo, mandar la salida (usualmente a consola) de un programa directamente a la impresora

```
out | lpr
```

## 16.3 E/S Basica

Hay un par de funciones que dan las facilidades básicas de E/S. Quizás las más comunes son: `getchar()` y `putchar()`. Están definidas y son usadas como sigue:

- `int getchar(void)` -- lee un caracter de stdin
- `int putchar(char ch)` -- escribe un caracter a stdout y regresa el caracter escrito.

```
main() {  
    int ch;  
  
    ch = getchar();  
    (void) putchar((char) ch); }  
}
```

Otras funciones relacionadas son:

```
int getc(FILE *flujo);  
int putc(char ch, FILE *flujo);
```

## 16.4 E/S formateada

Se han visto ya algunos ejemplos de como C usa la E/S formateada. En esta sección se revisarán con más detalle.

### 16.4.1 printf

El prototipo de la función esta definido como:

```
int printf( const char *formato, lista arg ...);
```

que muestra en stdout la lista de argumentos de acuerdo al formato especificado. La función devuelve el número de caracteres impresos.

La cadena de formateo tiene dos tipos de objetos:

- *caracteres ordinarios* -- estos son copiados a la salida.
- *especificadores de conversión* -- precedidos por % y listados en la tabla [16.1](#).

**Tabla 16.1:** Caracteres de format printf/scanf

<i>Especificador (%)</i>	<i>Tipo</i>	<i>Resultado</i>
c	char	un sólo caracter
i,d	int	número base diez
o	int	número base ocho
x,X	int	número base dieciseis
		notación minús/mayús
u	int	entero sin signo
s	char *	impresión de cadena
		terminada por nulo
f	double/float	formato -m.ddd ...
e,E	"	Notación Científica
		-1.23e002
g,G	"	e ó f la que sea
		más compacta
%	-	caracter %

Entre el % y el caracter de formato se puede poner:

- - signo menos para justificar a la izquierda.
- *número entero* para el ancho del campo.
- *m.d* en donde *m* es el ancho del campo, y *d* es la precisión de dígitos después del punto decimal o el número de caracteres de una cadena.

Por lo tanto:

```
printf("%-2.3f\n", 17.23478);
```

la salida en pantalla será:

```
17.235
```

y

```
printf("VAT=17.5%\n");
```

genera la siguiente salida:

```
VAT=17.5%
```

## 16.4.2 scanf

La función esta definida como sigue:

```
int scanf( const char *formato, lista arg ...);
```

Lee de la entrada estándar (stdin) y coloca la entrada en la dirección de las variables indicadas en *lista args*. Regresa el número de caracteres leídos.

La cadena de control de formateo es similar a la de `printf`.

Importante: se requiere la *dirección de la variable* o un apuntador con `scanf`.

Por ejemplo:

```
scanf("%d", &i);
```

Para el caso de un arreglo o cadena sólo se requiere el nombre del mismo para poder usar `scanf` ya que corresponde al inicio de la dirección de la cadena.

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

## 16.5 Archivos

Los archivos son la forma más común de los flujos.

Lo primero que se debe hacer es *abrir* el archivo. La función `fopen()` hace lo siguiente:

```
FILE *fopen(const char *nomb, const char *modo);
```

`fopen` regresa un apuntador a un `FILE`. En la cadena `nomb` se pone el nombre y la trayectoria del archivo que se desea acceder. La cadena `modo` controla el tipo de acceso. Si un archivo no puede ser accesado por alguna razón un apuntador `NULL` es devuelto.

Los modos son:

- `"r"` lectura;
- `"w"` escritura; y
- `"a"`

Para abrir un archivo se debe tener un flujo (apuntador tipo archivo) que apunte a la estructura `FILE`.

Por lo tanto, para abrir un archivo denominado *miarch.dat* para lectura haremos algo como lo siguiente;

```
FILE *flujo; /* Se declara un flujo */
flujo = fopen("miarch.dat", "r");
```

es una buena práctica revisar si un archivo se pudo abrir correctamente

```
if ( (flujo = fopen("miarch.dat", "r")) == NULL )
{
    printf("No se pudo abrir %s\n", "miarch.dat");
    exit(1);
}
.....
```

## 16.5.1 Lectura y escritura de archivos

Las funciones `fprintf` y `fscanf` son comúnmente empleadas para acceder archivos.

```
int fprintf(FILE *flujo, const char *formato, args ... );
int fscanf(FILE *flujo, const char *formato, args ... );
```

Las funciones son similares a `printf` y `scanf` excepto que los datos son leídos desde el *flujo*, el cual deberá ser abierto con `fopen()`.

El apuntador al flujo es automáticamente incrementado con *todas* las funciones de lectura y escritura. Por lo tanto, no se debe preocupar en hacer lo anterior.

```
char *cadena[80];
FILE *flujo;

if ( (flujo = fopen( ... )) != NULL)
    fscanf(flujo, "%s", cadena);
```

Otras funciones para archivos son:

```
int getc(FILE *flujo)                int fgetc(FILE *flujo)
int putc(char ch, FILE *s)          int fputc(char ch, FILE *s)
```

Estas son parecidas a `getchar` y `putc`.

`getc` esta definida como una macro del preprocesador en `stdio.h`. `fgetc` es una función de la biblioteca de C. Con ambas se consigue el mismo resultado.

Para el volcado de los datos de los flujos a disco, o bien, para disasociar un flujo a un archivo, haciendo previamente un volcado, usar:

```
int fflush(FILE *flujo);
int fclose(FILE *flujo);
```

También se puede tener acceso a los flujos predeterminados con `fprintf`, etc. Por ejemplo:

```
fprintf(stderr, "¡¡No se puede calcular!!\n");
fscanf(stdin, "%s", string);
```

## 16.6 `sprintf` y `sscanf`

Son parecidas a `fprintf` y `fscanf` excepto que escriben/leen una cadena.

```
int sprintf(char *cadena, char *formato, args ... )
int sscanf(char *cadena, char *formato, args ... )
```

Por ejemplo:

```
float tanque_lleno = 47.0;                /* litros */
float kilometros = 400;
char km_por_litro[80];

sprintf( km_por_litro, "Kilometros por litro = %2.3f", kilometros/tanque_lleno);
```

### 16.6.1 Petición del estado del flujo

Existen unas cuantas funciones útiles para conocer el estado de algún flujo y que tienen los prototipos siguientes:

```
int feof(FILE *flujo);
int ferror(FILE *flujo);
void clearerr(FILE *flujo);
int fileno(FILE *flujo);
```

- `feof()` devuelve verdadero si el flujo indicado esta en el fin del archivo. Por lo tanto para leer un flujo, `fp`, línea a línea se podría hacer algo como:

```
while ( !feof(fp) )
    fscanf(fp, "%s", linea);
```

- `ferror()` inspecciona el indicador de error para el flujo indicado, regresando verdadero si un

- error ha ocurrido.
- `clearerr()` limpia los indicadores de fin-de-fichero y error para el flujo indicado.
  - `fileno()` examina el argumento flujo y devuelve su descriptor de fichero, como un entero.

## 16.7 E/S de bajo nivel o sin almacenamiento intermedio

Esta forma de E/S es sin buffer -cada requerimiento de lectura/escritura genera un acceso al disco (o dispositivo) directamente para traer/poner un determinado número de bytes.

No hay facilidades de formateo -ya que se están manipulando bytes de información.

Lo anterior significa que se están usando archivos binarios (y no de texto).

En vez de manejar apuntadores de archivos, se emplea un *manejador de archivo de bajo nivel o descriptor de archivo*, el cual da un entero único para identificar cada archivo.

Para abrir un archivo usar:

```
int open(char* nomb, int flag);
```

que regresa un descriptor de archivo ó -1 si falla.

`flag` controla el acceso al archivo y tiene los siguientes macros definidas en `fcntl.h`:

- `O_APPEND` el archivo se abrirá en modo de sólo añadir
- `O_CREAT` si el archivo no existe, será creado.
- `O_EXCL` abrir en forma exclusiva.
- `O_RDONLY` sólo lectura
- `O_RDWR` lectura y escritura
- `O_WRONLY` sólo escritura

para ver otras opciones usar `man`.

La función:

```
int creat(char* nomb, int perms);
```

puede también ser usada para crear un archivo.

Otras funciones son:

```
int close(int fd);  
int read(int fd, char *buffer, unsigned longitud);  
int write(int fd, char *buffer, unsigned longitud);
```

que pueden ser usadas para cerrar un archivo y leer/escribir un determinado número de bytes de la memoria/hacia un archivo en la localidad de memoria indicada por `buffer`.

La función `sizeof()` es comúnmente usada para indicar la longitud.

Las funciones `read` y `write` regresan el número de bytes leídos/escritos o -1 si fallan.

Se tiene a continuación dos aplicaciones que usan algunas de las funciones indicadas:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main(int argc, char **argv)
{
    float buffer[]={23.34,2.34,1112.33};
    int df;
    int bytes_esc;
    int num_flot;

    /* Primeramente se crea el archivo */
    if ( (df = open(argv[1], O_CREAT, S_IRUSR | S_IWUSR) ) == -1)
    { /* Error, archivo no abierto */
        perror("Archivo datos, apertura");
        exit(1);
    }
    else
        printf("Descriptor de archivo %d\n",df);

    /* Despues se abre para solamente escribir */
    if ( (df = open(argv[1], O_WRONLY) ) == -1)
    { /* Error, archivo no abierto */
        perror("Archivo datos, apertura");
        exit(1);
    }
    else
        printf("Descriptor de archivo %d\n",df);

    /* En primer lugar se escribe el número de flotantes que seran escritos
*/
    num_flot = 3;
    if ( (bytes_esc = write(df, &num_flot, sizeof(int)) ) == -1)
    { /* Error en la escritura */
        perror("Archivo datos, escritura");
        exit(1);
    }
    else
        printf("Escritos %d bytes\n",bytes_esc);

    /* Se escribe el arreglo de flotantes */
    if ( (bytes_esc = write(df, buffer, num_flot*sizeof(float)) ) == -1)
    { /* Error en la escritura */
        perror("Archivo datos, escritura");
        exit(1);
    }
    else
        printf("Escritos %d bytes\n",bytes_esc);

    close(df);
}
```

Ejemplo de lectura del ejemplo anterior:

```

/* Este programa lee una lista de flotantes de un archivo binario. */
/* El primer byte del archivo es un entero indicando cuantos      */
/* flotantes hay en el archivo. Los flotantes estan despues del  */
/* entero, el nombre del archivo se da en la linea de comandos.   */

#include <stdio.h>
#include <fcntl.h>

main(int argc, char **argv)
{
    float buffer[1000];
    int fd;
    int bytes_leidos;
    int num_flot;

    if ( (fd = open(argv[1], O_RDONLY)) == -1)
    { /* Error, archivo no abierto */
        perror("Archivo datos");
        exit(1);
    }

    if ( (bytes_leidos = read(fd, &num_flot, sizeof(int))) == -1)
    { /* Error en la lectura */
        exit(1);
    }

    if ( num_flot > 999 ) { /* arch muy grande */ exit(1); }

    if ( (bytes_leidos = read(fd, buffer, num_flot*sizeof(float))) == -1)
    { /* Error en la lectura */
        exit(1);
    }
}

```

## 16.8 Ejercicios

1. Escribir un programa para copiar un archivo en otro. Los 2 nombres de los archivos son dados como los primeros argumentos del programa.

Copiar bloques de 512 bytes cada vez.

Revisar que el programa:

- tenga dos argumentos o mostrar "El programa requiere dos argumentos";
- el primer nombre de archivo sea de lectura o mostrar "No se puede abrir archivo ... para lectura";
- que el segundo nombre del archivo sea de escritura o mostrar "No se puede abrir archivo ... para escritura".

2. Escribir un programa "ultimas" que muestre las últimas  $n$  líneas de un archivo de texto.  $n$  y el nombre del archivo deberán especificarse desde la línea de comandos. Por defecto  $n$  deberá ser 5. El programa deberá hacer el mejor uso del espacio de almacenamiento.
  3. Escribir un programa que compare dos archivos y muestre las líneas que difieran. Tip: buscar rutinas apropiadas para manejo de cadenas y manejo de archivos. El programa no deberá ser muy grande.
-



---

## Subsecciones

- [17.1 Funciones básicas para el manejo de cadenas](#)
    - [17.1.1 Búsqueda en cadenas](#)
  - [17.2 Prueba y conversión de caracteres <ctype.h>](#)
  - [17.3 Operaciones con la memoria <memory.h>](#)
  - [17.4 Ejercicios](#)
- 

# 17. Manejo de cadenas <string.h>

Recordando la presentación de arreglos hecha (capítulo 5) en donde las cadenas están definidas como un arreglo de caracteres o un apuntador a una porción de memoria conteniendo caracteres ASCII. Una cadena en C es una secuencia de cero o más caracteres seguidas por un caracter NULL o `\0`:



**Figura 17.1:** Representación de un arreglo.

Es importante preservar el caracter de terminación NULL, ya que con éste es como C define y maneja las longitudes de las cadenas. Todas las funciones de la biblioteca estándar de C lo requieren para una operación satisfactoria.

En general, aparte de algunas funciones *restringidas en longitud* (`strncat()`, `strncmp()` y `strncpy()`), al menos que se creen cadenas a mano, no se deberán encontrar problemas. Se deberán usar las funciones para manejo de cadenas y no tratar de manipular las cadenas en forma manual desmantelando y ensamblando cadenas.

## 17.1 Funciones básicas para el manejo de cadenas

Todas las funciones para manejo de cadenas tienen su prototipo en:

```
#include <string.h>
```

Las funciones más comunes son descritas a continuación:

- `char *strcpy(const char *dest, const char *orig)` -- Copia la cadena de caracteres apuntada por `orig` (incluyendo el carácter terminador `'\0'`) al vector apuntado por `dest`. Las

cadenas no deben solaparse, y la de destino, debe ser suficientemente grande como para alojar la copia.

- `int strcmp(const char *s1, const char *s2)` -- Compara las dos cadenas de caracteres `s1` y `s2`. Devuelve un entero menor, igual o mayor que cero si se encuentra que `s1` es, respectivamente, menor que, igual a, o mayor que `s2`.
- `char *strerror(int errnum)` -- Devuelve un mensaje de error que corresponde a un número de error.
- `int strlen(const char *s)` -- Calcula la longitud de la cadena de caracteres.
- `char *strncat(char *s1, const char *s2, size_t n)` -- Agrega `n` caracteres de `s2` a `s1`.
- `int strncmp(const char *s1, char *s2, size_t n)` -- Compara los primeros `n` caracteres de dos cadenas.
- `char *strncpy(const char *s1, const char *s2, size_t n)` -- Copia los primeros `n` caracteres de `s2` a `s1`.
- `strcasecmp(const char *s1, const char *s2)` -- versión que ignora si son mayúsculas o minúsculas de `strcmp()`.
- `strncasecmp(const char *s1, const char *s2, size_t n)` -- versión insensible a mayúsculas o minúsculas de `strncmp()` que compara los primeros `n` caracteres de `s1`.

El uso de muchas funciones es directo, por ejemplo:

```
char *s1 = "Hola";
char *s2;
int longitud;

longitud = strlen("Hola");      /* long = 4 */
(void) strcpy(s2, s1);
```

Observar que tanto `strcat()` y `strcpy()` regresan una copia de su primer argumento, el cual es el arreglo destino. Observar también que orden de los argumentos es *arreglo destino* seguido por *arreglo fuente* lo cual a veces es una situación para hacerlo incorrectamente.

La función `strcmp()` compara lexicográficamente las dos cadenas y regresa:

- *Menor que cero* -- si `s1` es léxicamente menor que `s2`;
- *Cero* -- si `s1` y `s2` son léxicamente iguales;
- *Mayor que cero* -- si `s1` es léxicamente mayor que `s2`;

Las funciones de copiado `strncat()`, `strncmp()` y `strncpy()` son versiones más restringidas que sus contrapartes más generales. Realizan una tarea similar, pero solamente para los primeros `n` caracteres. Observar que el caracter de terminación NULL podría ser violado cuando se usa estas funciones, por ejemplo:

```
char *s1 = "Hola";
char *s2 = 2;
int longitud = 2;

(void) strncpy(s2, s1, longitud); /* s2 = "Ho" */
```

donde `s2` no tiene el terminador NULL.

## 17.1.1 Búsqueda en cadenas

La biblioteca también proporciona varias funciones de búsqueda en cadenas.

- `char *strchr(const char *s, int c)` -- Devuelve un puntero a la primera ocurrencia del carácter `c` en la cadena de caracteres `s`.
- `char *strrchr(const char *s, int c)` -- Encuentra la última ocurrencia del carácter `c` en la cadena.
- `char *strstr(const char *s1, const char *s2)` -- Localiza la primera ocurrencia de la cadena `s2` en la cadena `s1`.
- `char *strpbrk(const char *s1, const char *s2)` -- Regresa un apuntador a la primera ocurrencia en la cadena `s1` de cualquier carácter de la cadena `s2`, o un apuntador nulo si no hay un carácter de `s2` que exista en `s1`.
- `size_t strspn(const char *s1, const char *s2)` -- Calcula la longitud del segmento inicial de `s1` que consta únicamente de caracteres en `s2`.
- `size_t strcspn(const char *s1, const char *s2)` -- Regresa el número de caracteres al principio de `s1` que no coinciden con `s2`.
- `char *strtok(char *s1, const char *s2)` -- Divide la cadena apuntada a `s1` en una secuencia de "tokens", cada uno de ellos está delimitado por uno o más caracteres de la cadena apuntada por `s2`.

Las funciones `strchr()` y `strrchr()` son las más simples de usar, por ejemplo:

```
char *s1 = "Hola";
char *resp;

resp = strchr(s1, 'l');
```

Después de la ejecución, `resp` apunta a la localidad `s1 + 2`.

La función `strpbrk()` es una función más general que busca la primera ocurrencia de cualquier grupo de caracteres, por ejemplo:

```
char *s1 = "Hola";
char *resp;

res = strpbrk(s1, "aeiou");
```

En este caso, `resp` apunta a la localidad `s1 + 1`, la localidad de la primera `o`.

La función `strstr()` regresa un apuntador a la cadena de búsqueda especificada o un apuntador nulo si la cadena no es encontrada. Si `s2` apunta a una cadena de longitud cero (esto es, la cadena ""), la función regres `s1`. Por ejemplo:

```
char *s1 = "Hola";
char *resp;

resp = strstr(s1, "la");
```

la cual tendrá `resp = s1 + 2`.

La función `strtok()` es un poco más complicada en cuanto a operación. Si el primer argumento

no es *NULL* entonces la función encuentra la posición de cualquiera de los caracteres del segundo argumento. Sin embargo, la posición es recordada y cualquier llamada subsecuente a `strtok()` iniciará en esa posición si en estas subsecuentes llamadas el primer argumento no es *NULL*. Por ejemplo, si deseamos dividir la cadena `s1` usando cada espacio e imprimir cada "token" en una nueva línea haríamos lo siguiente:

```
char s1[] = "Hola muchacho grande";
char *t1;

for ( t1 = strtok(s1, " ");
      t1 != NULL;
      t1 = strtok(NULL, " ") )
    printf("%s\n", t1);
```

Se emplea un ciclo `for` en una forma no regular de conteo:

- En la inicialización se llama a la función `strtok()` con la cadena `s1`.
- Se termina cuando `t1` es *NULL*.
- Se esta asignando tokens de `s1` a `t1` hasta la terminación llamando a `strtok()` con el primer argumento *NULL*.

## 17.2 Prueba y conversión de caracteres

### <ctype.h>

Una biblioteca relacionada `#include <ctype.h>` la cual contiene muchas funciones útiles para convertir y probar caracteres *individuales*.

Las funciones más comunes para revisar caracteres tienen los siguientes prototipos:

- `int isalnum(int c)` -- Verdad si `c` es alfanumérico.
- `int isalpha(int c)` -- Verdad si `c` es una letra.
- `int isascii(int c)` -- Verdad si `c` es ASCII.
- `int iscntrl(int c)` -- Verdad si `c` es un caracter de control.
- `int isdigit(int c)` -- Verdad si `c` es un dígito decimal.
- `int isgraph(int c)` -- Verdad si `c` es un caracter imprimible, exceptuando el espacio en blanco.
- `int islower(int c)` -- Verdad si `c` es una letra minúscula.
- `int isprint(int c)` -- Verdad si `c` es un caracter imprimible, incluyendo el espacio en blanco.
- `int ispunct(int c)` -- Verdad si `c` es un signo de puntuación.
- `int isspace(int c)` -- Verdad si `c` es un espacio
- `int isupper(int c)` -- Verdad si `c` es una letra mayúscula.
- `int isxdigit(int c)` -- Verdad si `c` es un dígito hexadecimal.

Las funciones para conversión de caracteres son:

- `int toascii(int c)` -- Convierte `c` a ASCII o un `unsigned char` de 7 bits, borrando los bits altos.
- `int tolower(int c)` -- Convierte la letra `c` a minúsculas, si es posible.
- `int toupper(int c)` -- Convierte la letra `c` a mayúsculas, si es posible.

El uso de estas funciones es directo y por lo tanto, no se dan ejemplos.

## 17.3 Operaciones con la memoria

### <memory.h>

Finalmente se verá un resumen de algunas funciones básicas de memoria. Sin embargo, no son funciones estrictamente de cadenas, pero tienen su prototipo en `#include <string.h>`:

- `void *memchr(void *s, int c, size_t n)` -- Busca un caracter en un buffer.
- `int memcmp(void *s1, void *s2, size_t n)` -- Compara dos buffers.
- `void *memcpy(void *dest, void *fuente, size_t n)` -- Copia un buffer dentro de otro.
- `void *memmove(void *dest, void *fuente, size_t n)` -- Mueve un número de bytes de un buffer a otro.
- `void *memset(void *s, int c, size_t n)` -- Pone todos los bytes de un buffer a un caracter dado.

El uso de estas funciones es directo y parecido a las operaciones de comparación de caracteres (excepto que la longitud exacta ( $n$ ) de todas las operaciones deberá ser indicada ya que no hay una forma propia de terminación).

Observar que en todos los casos bytes de memoria son copiados, por lo que la función `sizeof()` ayuda en estos casos, por ejemplo:

```
char fuente[TAM], dest[TAM];
int ifuente[TAM], idest[TAM];

memcpy(dest, fuente, TAM); /* Copia chars (bytes) OK */
memcpy(idest, ifuente, TAM*sizeof(int)); /* Copia arreglo de enteros */
```

La función `memmove()` se comporta de la misma forma que `memcpy()` excepto que las localidades de la fuente y el destino podrían traslaparse.

La función `memcmp()` es similar a `strcmp()` excepto que *unsigned bytes* son comparados y se devuelve cero si  $s1$  es menor que  $s2$ , etc.

## 17.4 Ejercicios

1. Escribir una función similar a `strlen` que pueda manejar cadenas sin terminador. Tip: se necesitará conocer y pasar la longitud de la cadena.
2. Escribir una función que regrese verdad, si una cadena de entrada es un palíndromo. Un palíndromo es una palabra que se lee igual de izquierda a derecha, o de derecha a izquierda. Por ejemplo, ANA.
3. Sugerir una posible implementación de la función `strtok()`:
  1. usando otras funciones de manejo de cadenas.
  2. desde los principios de apuntadores.¿Cómo se logra el almacenamiento de una cadena separada?
4. Escribir una función que convierta todos los caracteres de una cadena a mayúsculas.

5. Escribir un programa que invierta el contenido de la memoria en bytes. Es decir, si se tienen  $n$  bytes, el byte de memoria  $n$  se invierte con el byte 0, el byte  $n-1$  se intercambia con el byte 1, etc.
- 
-

---

## Subsecciones

- [18.1 Funciones para el manejo de directorios <unistd.h>](#)
    - [18.1.1 Búsqueda y ordenamiento de directorios: sys/types.h, sys/dir.h](#)
  - [18.2 Rutinas de manipulación de archivos: unistd.h, sys/types.h, sys/stat.h](#)
    - [18.2.1 Permisos de accesos a archivos](#)
      - [18.2.1.1 errno](#)
    - [18.2.2 Estado de un archivo](#)
    - [18.2.3 Manipulación de archivos: stdio.h, unistd.h](#)
    - [18.2.4 Creación de archivos temporales: <stdio.h>](#)
  - [18.3 Ejercicios](#)
- 

# 18. Acceso de Archivos y llamadas al sistema de directorios

Existen muchas utilerías que permiten manipular directorios y archivos. Los comandos `cd`, `ls`, `rm`, `cp`, `mkdir`, *etc.* son ejemplos que han sido revisados ya en el sistema operativo.

Se revisará en este capítulo como hacer las mismas tareas dentro de un programa en C.

## 18.1 Funciones para el manejo de directorios <unistd.h>

Para realizarlo se requiere llamar las funciones apropiadas para recorrer la jerarquía de directorios o preguntar sobre los contenidos de los directorios.

- `int chdir(char *path)` -- cambia al directorio indicado en la cadena `path`.
- `char *getcwd(char *path, size_t tam)` -- Obtiene la trayectoria completa, es decir, desde la raíz del directorio actual. `path` es un apuntador a una cadena donde la trayectoria será regresada. La función devuelve un apuntador a la cadena o `NULL` si un error ocurre.

Se tiene a continuación la emulación del comando `cd` de Unix con C:

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

#define TAM 80

main(int argc, char **argv)
```

```

{
    char cadena[TAM];

    if (argc < 2)
    {
        printf("Uso: %s <directorio> \n",argv[0]);
        exit(1);
    }

    if (chdir(argv[1]) != 0)
    {
        printf("Error en %s.\n",argv[0]);
        exit(1);
    }

    getcwd(cadena,TAM);

    printf("El directorio actual es %s\n",cadena);

}

```

## 18.1.1 Búsqueda y ordenamiento de directorios: sys/types.h,sys/dir.h

Dos funciones útiles (en plataformas BSD y aplicaciones no multi-*thread*) están disponibles:

```

int scandir(const char *dir, struct dirent **listanomb,
int (*select)(const struct dirent *),
int (*compar)(const struct dirent **, const struct dirent **))

```

Esta función rastrea el directorio `dir`, llamando `select()` en cada entrada de directorio. Las entradas para las que `select()` devuelve un valor distinto de cero se almacenan en cadenas que se asignan de la memoria con `malloc()`, ordenadas usando `qsort()` con la función de comparación `compar()`, y puestas en la matriz `listanomb`. Si `select` es `NULL`, se seleccionan todas las entradas.

```

int alphasort(const struct dirent **a, const struct dirent **b);

```

Puede ser usada como función de comparación para que la función `scandir()` ponga las entradas de directorio en orden alfabético.

A continuación se tiene una versión simple del comando de UNIX `ls`

```

#include <dirent.h>
#include <unistd.h>
#include <sys/param.h>
#include <stdio.h>

#define FALSO 0
#define VERDADERO !FALSO

extern int alphasort();

char trayectoria[MAXPATHLEN];

```

```

main()
{
    int contar,i;
    struct dirent **archivos;
    int selecc_arch();

    if ( getwd(trayectoria) == NULL )
    {
        printf("Error obteniendo la trayectoria actual\n");
        exit(0);
    }
    printf("Directorio de trabajo actual: %s\n",trayectoria);
    contar = scandir(trayectoria, &archivos, selecc_arch, alphasort);

    /* Si no se encontraron archivos */
    if (contar <= 0)
    {
        printf("No hay archivos en este direntorio\n");
        exit(0);
    }
    printf("Hay %d archivos.\n",contar);
    for (i=0; i<contar; ++i)
        printf("%s  ",archivos[i]->d_name);
    printf("\n"); /* flush buffer */
}

int selecc_arch(struct dirent *entry)
{
    if ((strcmp(entry->d_name, ".") == 0) ||
        (strcmp(entry->d_name, "..") == 0))
        return (FALSO);
    else
        return (VERDADERO);
}

```

scandir regresa los pseudodirectorios (.), (..) y también todos los archivos. En este ejemplo se filtran los pseudodirectorios para que no se muestren haciendo que la función regrese FALSO.

Los prototipos de las funciones scandir y alphasort tienen definiciones en la cabecera dirent.h

MAXPATHLEN esta definida en sys/param.h y la función getwd() en unistd.h.

Se puede ir más lejos que lo anterior y buscar por archivos particulares.

Reescribiendo la función selecc\_arch para que sólo muestre los archivos con los sufijos .c, .h y .o.

```

int selecc_arch(struct dirent *entry)
{
    char *ptr;

    if ((strcmp(entry->d_name, ".") == 0) ||
        (strcmp(entry->d_name, "..") == 0))
        return (FALSO);

    /* Revisar las extensiones de los archivos */
    ptr = rindex(entry->d_name, '.'); /* Probar que tenga un punto */

    if ( (ptr != NULL) &&
        ( (strcmp(ptr, ".c") == 0)
          || (strcmp(ptr, ".h") == 0)
          || (strcmp(ptr, ".o") == 0) ) ) )
        return (VERDADERO);
    else
        return (FALSO);
}

```

```

        || (strcmp(ptr, ".o") == 0) )
    )
    return (VERDADERO);
else
    return(FALSO);
}

```

La función `rindex()` es una función para manejo de cadenas que regresa un apuntador a la última ocurrencia del carácter `c` en la cadena `s`, o un apuntador `NULL` si `c` no ocurre en la cadena. (`index()` es una función similar pero asigna un apuntador a la primera ocurrencia.)

## 18.2 Rutinas de manipulación de archivos: unistd.h, sys/types.h, sys/stat.h

Existen varias llamadas al sistema que pueden ser aplicadas directamente a los archivos guardados en un directorio.

### 18.2.1 Permisos de accesos a archivos

La función `int access(const char *trayectoria, int modo);` -- determina los permisos de usuario para un fichero, de acuerdo con `modo`, que esta definido en `#include <unistd.h>`, los cuales pueden ser:

- `R_OK` -prueba el permiso de lectura.
- `W_OK` -prueba el permiso de escritura.
- `X_OK` -prueba el permiso de ejecución o búsqueda.
- `F_OK` -prueba si se permite la comprobación de la existencia del fichero.

La función `access()` regresa: 0 si ha habido éxito, o -1 en caso de falla y a `errno` se le asigna un valor adecuado. Ver las páginas del `man` para ver la lista de errores.

#### 18.2.1.1 errno

Algunas llamadas al sistema (y algunas funciones de biblioteca) dan un valor al entero `errno` para indicar que ha habido un error. Esta es una variable especial del sistema.

Para usar `errno` en un programa de C, debe ser declarado de la siguiente forma:

```
extern int errno;
```

Esta puede ser manualmente puesto dentro de un programa en C, de otra forma simplemente retiene el último valor.

La función `int chmod(const char *trayectoria, mode_t modo);` cambia el modo del archivo dado mediante `trayectoria` para un modo dado.

`chmod()` devuelve 0 en caso de éxito y -1 en caso de error además se asigna a la variable `errno` un valor adecuado (revisar las páginas de `man` para ver los tipos de errores)

## 18.2.2 Estado de un archivo

Se tienen dos funciones útiles para conocer el estado actual de un archivo. Por ejemplo se puede saber que tan grande es un archivo (`st_size`), cuando fue creado (`st_ctime`), etc (ver la definición de la estructura abajo). Las dos funciones tienen sus prototipos en `<sys/stat.h>`

- `int stat(const char *nomb_arch, struct stat *buf)` -- obtiene información acerca del archivo apuntado por `nomb_arch`. No se requieren permisos de lectura, escritura o ejecución, pero todos los directorios listados en `nomb_arch` deberán estar disponibles.
- `int fstat(int desarch, struct stat *buf)` -- obtiene la misma información que el anterior, pero sólo el archivo abierto apuntado por `desarch` (tal y como lo devuelve `open()`) es examinado en lugar de `nomb_arch`.

Las funciones `stat()` y `fstat()` regresan 0 en caso éxito, -1 si hubo error y `errno` es actualizado apropiadamente.

`buf` es un apuntador a la estructura `stat` en la cual la información es colocada. La estructura `stat` esta definida en include `<sys/stat.h>`, como sigue:

```
struct stat
{
    dev_t          st_dev;          /* dispositivo */
    ino_t          st_ino;         /* inodo */
    mode_t        st_mode;        /* proteccion */
    nlink_t       st_nlink;       /* numero de enlaces fisicos */
    uid_t         st_uid;         /* ID del usuario propietario */
    gid_t         st_gid;         /* ID del grupo propietario */
    dev_t         st_rdev;        /* tipo dispositivo (si es
                                dispositivo inodo) */
    off_t         st_size;        /* tamaño total, en bytes */
    unsigned long st_blksize;     /* tamaño de bloque para el
                                sistema de ficheros de E/S */
    unsigned long st_blocks;     /* numero de bloques asignados */
    time_t        st_atime;       /* hora ultimo acceso */
    time_t        st_mtime;       /* hora ultima modificacion */
    time_t        st_ctime;       /* hora ultimo cambio */
};
```

Se muestra un ejemplo que hace uso de la función `stat`:

```
#include <stdio.h>
#include <sys/stat.h> /* Para la estructura stat */
#include <unistd.h>

main(int argc, char **argv)
{
    struct stat buf;

    printf("%s\n", argv[0]);

    if ( stat(argv[0], &buf) == -1 )
    {
```

```

        perror(argv[0]);
        exit(-1);
    }
    else
    {
        printf("Tamaño del archivo %s %d bytes.\n",argv[0],buf.st_size);
    }
}

```

## 18.2.3 Manipulación de archivos: `stdio.h`, `unistd.h`

Existen algunas funciones para borrar y renombrar archivos. Quizás la forma más común es usando las funciones de `stdio.h`:

```

int remove(const char *pathname);
int rename(const char *viejo, const char *nuevo);

```

Dos llamadas al sistema (definidas en `unistd.h`) las cuales son actualmente usadas por las funciones `remove()` y `rename()` también existen, pero son probablemente más difíciles de recordar, al menos que se este suficientemente familiarizado con UNIX.

- *int unlink(const char \*pathname);* -- borra un nombre del sistema de archivos. Si dicho nombre era el último enlace a un archivo, y ningún proceso tiene el archivo abierto, el fichero es borrado y el espacio que ocupaba vuelve a estar disponible. Regresa 0 en caso de éxito, -1 en caso de error y pone un valor en `errno` para indicarlo.
- *int link(const char \*oldpath, const char \*newpath);* -- crea un nuevo enlace (también conocido como enlace físico a un archivo existente).

Las funciones `stat()` y `fstat()` regresan 0 en caso éxito, -1 si hubo error y pone la variable `errno` con algún valor indicando el tipo de error.

## 18.2.4 Creación de archivos temporales: `<stdio.h>`

Los programas con frecuencia necesitan crear archivos sólo durante la vida de un programa. Existen dos funciones convenientes (además de algunas variantes) para la realización de la tarea mencionada. El manejo (borrado de archivos, etc.) es tomado con cuidado por el sistema operativo.

La función `FILE *tmpfile (void);` crea un archivo temporal (en modo de lectura/escritura binaria) y abre el correspondiente flujo. El archivo se borrará automáticamente cuando el programa termine.

La función `char *tmpnam(char *s);` crea un nombre único para un archivo temporal usando el prefijo de trayectoria `P_tmpdir` definido en `<stdio.h>`.

## 18.3 Ejercicios

1. Escribir un programa en C para simular el comando `ls -l` de UNIX que muestre todos los

- archivos del directorio actual, sus permisos, tamaño, etc.
2. Escribir un programa para mostrar las líneas de un archivo que contenga una palabra dada como argumento al programa, es decir, una versión sencilla de la utilidad `grep` de UNIX.
  3. Escribir un programa para mostrar una lista de archivos dados como argumentos, parando cada 20 líneas hasta que una tecla sea presionada (una versión simple de la utilidad `more` de UNIX).
  4. Escribir un programa que liste todos los archivos del directorio actual y todos los archivos en los subsecuentes directorios.
- 
-

---

## Subsecciones

- [19.1 Funciones básicas para el tiempo](#)
  - [19.2 Ejemplos de aplicaciones de funciones del tiempo.](#)
    - [19.2.1 Ejemplo 1: Tiempo \(en segundos\) para hacer algún cálculo.](#)
    - [19.2.2 Ejemplo 2: Inicializar la semilla de un número aleatorio.](#)
  - [19.3 Ejercicios](#)
- 

# 19. Funciones para el tiempo

En este capítulo se revisará como se puede usar el reloj con llamadas al sistema UNIX.

Existen muchas funciones para el tiempo que las que serán consideradas aquí -- usar las páginas de man y los listados de la biblioteca estándar de funciones para más detalles.

El uso más común de las funciones de tiempo son:

- conocer el tiempo,
- tomar el tiempo a programas y funciones y
- poner valores a semillas.

## 19.1 Funciones básicas para el tiempo

Algunas prototipos de las funciones básicas para el tiempo son las siguientes:

- *time\_t time(time\_t \*t)* -- devuelve el tiempo transcurrido, medido en segundos desde "la época" 0 horas, 0 minutos, 0 segundos, tiempo universal coordinado (GMT) del 1<sup>o</sup> de enero de 1970. Esta medida se llama el "tiempo de calendario". Si *t* no es nulo, el valor devuelto también se guarda en la zona de memoria a la que apunta *t*. En caso de error, se devuelve  $((\text{time\_t}) - 1)$  y se asigna a la variable `errno` un valor apropiado.
- *ftime(struct timeb \*pt)* -- devuelve la hora y la fecha actuales en *pt*, que esta declarada en `<sys/timeb.h>` como sigue:

```
struct timeb
{
    time_t time;           /* Segundos desde epoca, igual que
`time'. */
    unsigned short int millitm; /* milisegundos adicionales.
*/
    short int timezone;     /* Tiempo local medido en minutos
oeste de GMT.*/
    short int dstflag;     /* No cero si se usa horario de verano
*/
```

```
};
```

En caso de éxito, se devuelve el tiempo transcurrido en segundos desde la época. En caso de error, se devuelve  $((time\_t) - 1)$  y se asigna a la variable `errno` un valor apropiado.

- `char *ctime(time_t *timep)` -- toma un argumento de tipo `time_t` (long integer) que representa el tiempo de calendario y lo convierte a una cadena de 26 caracteres de la forma producida por la función `asctime()`. En primer lugar descompone los segundos a una estructura `tm` llamando a `localtime()`, y entonces llama a `asctime()` para convertir la estructura `tm` a una cadena.
- `char *asctime(const struct tm *timeptr)` -- convierte un valor de tiempo contenido en una estructura `tm` a una cadena de 26 caracteres de la forma:

*ddd mmm dd hh:mm:ss aaaa*

La función `asctime()` regresa un apuntador a la cadena.

## 19.2 Ejemplos de aplicaciones de funciones del tiempo.

Como se menciono previamente, básicamente existen tres posibles usos de las funciones de tiempo.

### 19.2.1 Ejemplo 1: Tiempo (en segundos) para hacer algún cálculo.

Este es un ejemplo sencillo que ilustra las llamadas a la función tiempo en distintos momentos:

```
/* timer.c */
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{
    int i, j;
    time_t t1, t2;

    (void) time(&t1);
    for (i=1; i<=300; ++i)
    {
        printf("%d %d %d\n", i, i*i, i*i*i );
        for(j=0; j<1000000; j++); /* Un pequeño retardo */
    }
    (void) time(&t2);
    printf("\n Tiempo para hacer 300 cuadrados y cubos = %d segundos\n",
        (int) t2-t1);
}
```

## 19.2.2 Ejemplo 2: Inicializar la semilla de un número aleatorio.

Se ha visto un ejemplo similar previamente, en esta ocasión se usa la función `lrand48()` para generar una secuencia de números:

```
/* random.c */
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{
    int i;
    time_t t1;

    (void) time(&t1);
    srand48((long) t1);
    /* usar time en segundos para poner la semilla */
    printf("5 numeros aleatorios (semilla = %d):\n", (int) t1);
    for (i=0; i<5; ++i)
        printf("%d ", lrand48());
    printf("\n\n");
}
```

La función `lrand48()` devuelve enteros largos no negativos distribuidos uniformemente entre 0 y  $2^{31}$ .

Una función similar es la función `drand48` que regresa números de doble precisión en el rango  $[0.0, 1.0)$ .

`srand48()` pone el valor de la semilla para estos generadores de números aleatorios. Es importante tener diferentes semillas cuando se llame a las funciones de otra forma el mismo conjunto números pseudo-aleatorios sera generados. La función `time()` siempre da una semilla única (siempre y cuando haya transcurrido por lo menos 1 segundo).

## 19.3 Ejercicios

1. Escribir un programa en C que mida el tiempo de un fragmento de código en milisegundos.
  2. Escribir un programa en C para producir una serie de números aleatorios de punto flotante en los rangos a) 0.0 - 1.0, b) 0.0 - n, donde n es cualquier valor flotante. La semilla deberá estar puesta para que se garantice una secuencia única.
- 
-

---

## Subsecciones

- [20.1 Ejecutando comandos de UNIX desde C](#)
  - [20.2 `execl\(\)`](#)
  - [20.3 `fork\(\)`](#)
  - [20.4 `wait\(\)`](#)
  - [20.5 `exit\(\)`](#)
  - [20.6 Ejercicios](#)
- 

# 20. Control de procesos: `<stdlib.h>`, `<unistd.h>`

Un *proceso* es básicamente un único programa ejecutándose. Este podría ser un programa del "sistema" (por ejemplo, `login`, `csch`, `update`, etc). un programa iniciado por el usuario (`gimp` o alguno hecho por el usuario).

Cuando UNIX ejecuta un proceso le asigna a cada proceso un número único -- *process ID* o *pid*.

El comando `ps` de UNIX lista todos los procesos que se están ejecutando en la máquina, listando también el *pid*.

La función de C `int getpid()` regresará el *pid* de un proceso que llame a esta función.

Un programa usualmente ejecuta un sólo proceso. Sin embargo después se verá como se puede hacer que un programa corra como varios procesos separados comunicados.

## 20.1 Ejecutando comandos de UNIX desde C

Se pueden ejecutar comandos desde un programa de C como si se estuviera en la línea de comandos de UNIX usando la función `system()`. NOTA: se puede ahorrar bastante tiempo y confusión en vez de ejecutar otros programas, scripts, etc. para hacer las tareas.

`int system(char *mandato)` -- donde *mandato* puede ser el nombre de una utilidad de UNIX, un shell ejecutable o un programa del usuario. La función regresa el status de salida del shell. La función tiene su prototipo en `<stdlib.h>`

Ejemplo: llamada del comando `ls` desde un programa

```
main()
{
    printf("Archivos en el directorio son:\n");
    system("ls -l");
}
```

La función `system` es una llamada que esta construida de otras 3 llamadas del sistema: `execl()`, `wait()` y `fork()` (las cuales tienen su prototipo en `<unistd.h>`).

## 20.2 `execl()`

La función `execl` tiene otras 5 funciones relacionadas -- ver las páginas de man.

La función `execl` realiza la *ejecución (execute)* y *sale (leave)*, es decir que un proceso será ejecutado y entonces terminado por `execl`.

Esta definida como:

```
execl (const char *camino, const char *arg0, ..., char *argn, 0);
```

El último parámetro deberá ser siempre `0`. Este es un terminador `NULO`. Como la lista de argumentos sea variable se debe indicar de alguna forma a C cuando se termine la lista. El terminador nulo hace lo anterior.

El apuntador `camino` indica el nombre de un archivo que contiene el comando que será ejecutado, `arg0` apunta a una cadena que es el mismo nombre (o al menos su último componente).

`arg1, ... , argn` son apuntadores a los argumentos para el comando y el `0` solamente indica el fin de la lista de argumentos variables.

Por lo tanto nuestro ejemplo queda de la siguiente forma:

```
#include <unistd.h>

main()
{
    printf("Los archivos en el directorio son:\n");
    execl("/bin/ls", "ls", "-l", 0);
    printf("!!! Esto no se ejecuta !!!\n");
}
```

## 20.3 `fork()`

La función `int fork()` cambia un proceso único en 2 procesos idénticos, conocidos como el *padre (parent)* y el *hijo (child)*. En caso de éxito, `fork()` regresa `0` al proceso hijo y regresa el identificador del proceso hijo al proceso padre. En caso de falla, `fork()` regresa `-1` al proceso padre, pone un valor a `errno`, y no se crea un proceso hijo.

El proceso hijo tendrá su propio identificador único (PID).

El siguiente programa ilustra un ejemplo sencillo del uso de `fork`, donde dos copias son hechas y se ejecutan juntas (multitarea).

```
main()
```

```

{
    int valor_regr=0;

    printf("Bifurcando el proceso\n");
    valor_regr=fork();
    printf("El id del proceso es %d y el valor regresado es %d\n",
           getpid(), valor_regr);
    execl("/bin/ls", "ls", "-l", 0);
    printf("Esta linea no es impresa\n");
}

```

La salida podría ser como la siguiente:

```

Bifurcando el proceso
El id del proceso es 2662 y el valor regresado es 2663
El id del proceso es 2663 y el valor regresado es 0

```

Los procesos tienen un único identificador, el cual será diferente en cada ejecución.

Es imposible indicar con antelación cual proceso obtendrá el CPU. Cuando un proceso es duplicado en 2 procesos se puede detectar fácilmente (en cada proceso) si el proceso es el hijo o el padre ya que `fork()` regresa **0** para el hijo. Se pueden atrapar cualquier error, ya que `fork()` regresa un **-1**, por ejemplo:

```

int pid;                /* identificador del proceso */

pid = fork();
if ( pid < 0 )
{
    printf(";; No se pudo duplicar !!\n");
    exit(1);
}

if ( pid == 0 )
{
    /* Proceso hijo */
    .....
}
else
{
    /* Proceso padre */
    .....
}

```

## 20.4 wait()

La función `int wait() (int *status)` forzará a un proceso padre para que espere a un proceso hijo que se detenga o termine. La función regresa el PID del hijo o **-1** en caso de error. El estado de la salida del hijo es regresado en `status`.

## 20.5 exit ()

La función `void exit(int status)` termina el proceso que llama a esta función y regresa en la salida el valor de `status`. Tanto UNIX y los programas bifurcados de C pueden leer el valor de `status`.

Por convención, un estado de `0` significa *terminación normal* y cualquier otro indica un error o un evento no usual. Muchas llamadas de la biblioteca estándar tienen errores definidos en la cabecera de archivo `sys/stat.h`. Se puede fácilmente derivar su propia convención.

Un ejemplo completo de un programa de bifurcación se muestra a continuación:

```
/*
fork.c - ejemplo de un programa de bifurcación
El programa pide el ingreso de comandos de UNIX que son dejados en una
cadena. La cadena es entonces "analizada" encontrando blancos, etc
Cada comando y sus correspondientes argumentos son puestos en
un arreglo de argumentos, execvp es llamada para ejecutar estos comandos
en un proceso hijo generado por fork()
*/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
    char buf[1024];
    char *args[64];

    for (;;)
    {
        /*
        * Pide y lee un comando.
        */
        printf("Comando: ");

        if (gets(buf) == NULL)
        {
            printf("\n");
            exit(0);
        }

        /*
        * Dividir la cadena en argumentos.
        */
        parse(buf, args);

        /*
        * Ejecutar el comando.
        */
        ejecutar(args);
    }
}

/*
* parse--divide el comando que esta en buf
* en argumentos.
*/
parse(char *buf, char **args)
{
```

```

while (*buf != (char) NULL)
{
    /*
    * Quitar blancos. Usar nulos, para que
    * el argumento previo sea terminado
    * automaticamente.
    */
    while ( (*buf == ' ') || (*buf == '\t') )
        *buf++ = (char) NULL;

    /*
    * Guardar los argumentos
    */
    *args++ = buf;

    /*
    * Brincar sobre los argumentos.
    */
    while ((*buf != (char) NULL) && (*buf != ' ') && (*buf != '\t'))
        buf++;
}

*args = (char) NULL;
}

/*
* ejecutar--genera un proceso hijo y ejecuta
* el programa.
*/
ejecutar(char **args)
{
    int pid, status;

    /*
    * Obtener un proceso hijo.
    */
    if ( (pid = fork()) < 0 )
    {
        perror("fork");
        exit(1);

        /* NOTA: perror() genera un mensaje de error breve en la
        * salida de errores describiendo el ultimo error encontrado
        * durante una llamada al sistema o funcion de la biblioteca.
        */
    }

    /*
    * El proceso hijo ejecuta el codigo dentro del if.
    */
    if (pid == 0)
    {
        execvp(*args, args);
        perror(*args);
        exit(1);

        /* NOTA: las versiones execv() y execvp() de execl() son utiles cuando
        el numero de argumentos es desconocido previamente.
        Los argumentos para execv() y execvp() son el nombre del archivo
        que sera ejecutado y un vector de cadenas que contienen los
        argumentos.
        El ultimo argumento de cadema debera ser un apuntador a 0 (NULL)
        execlp() y execvp() son llamados con los mismos argumentos que

```

```
execl() y execv(), pero duplican las acciones del shell en
la busqueda de un archivo ejecutable en un lista de directorios.
La lista de directorios es obtenida del ambiente.
*/
}

/*
* El padre ejecuta el wait.
*/
while (wait(&status) != pid)
    /* vacio */ ;
}
```

## 20.6 Ejercicios

1. Crear un programa en C en donde se use la función `popen()` para entubar la salida del comando `rwsh` de UNIX en el comando `more` de UNIX.
- 
-

---

## Subsecciones

- [21.1 Archivos Cabezera](#)
- [21.2 Variables y Funciones Externas](#)
  - [21.2.1 Alcance de las variables externas](#)
- [21.3 Ventajas de Usar Varios Archivos](#)
- [21.4 Como dividir un programa en varios archivos](#)
- [21.5 Organización de los Datos en cada Archivo](#)
- [21.6 La utilería Make](#)
  - [21.6.1 Programando Make](#)
- [21.7 Creación de un Archivo Make \(Makefile\)](#)
- [21.8 Uso de macros con Make](#)
- [21.9 Ejecución de Make](#)

---

# 21. Compilación de Programas con Archivos Múltiples

En este capítulo se revisa los aspectos teóricos y prácticos que necesitan ser considerados cuando son escritos programas grandes.

Cuando se escriben programas grandes se deberá programar en módulos. Estos serán archivos fuentes separados. La función `main()` deberá estar en un archivo, por ejemplo en `main.c`, y los otros archivos tendrán otras funciones.

Se puede crear una biblioteca propia de funciones escribiendo una *suite* de subrutinas en uno o más módulos. De hecho los módulos pueden ser compartidos entre muchos programas simplemente incluyendo los módulos al compilar como se verá a continuación.

Se tiene varias ventajas si los programas son escritos de esta forma:

- los módulos de forma natural se dividirán en grupos comunes de funciones.
- se puede compilar cada módulos separadamente y ligarlo con los módulos ya compilados.
- las utilerías tales como `make` nos ayudan a mantener sistemas grandes.

# 21.1 Archivos Cabezera

Si se adopta el modelo modular entonces se querrá tener para cada módulo las definiciones de las variables, los prototipos de las funciones, etc. Sin embargo, ¿qué sucede si varios módulos necesitan compartir tales definiciones? En tal caso, lo mejor es centralizar las definiciones en un archivo, y compartir el archivo entre los módulos. Tal archivo es usualmente llamado un *archivo cabecera*.

Por convención estos archivos tienen el sufijo `.h`

Se han revisado ya algunos archivos cabecera de la biblioteca estándar, por ejemplo:

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

Se pueden definir los propios archivos cabecera y se pueden incluir en el programa como se muestra enseguida:

```
#include "mi_cabecera.h"
```

Los archivos cabecera por lo general sólo contienen definiciones de tipos de datos, prototipos de funciones y comandos del preprocesador de C.

Considerar el siguiente programa de ejemplo:

## *main.c*

```
/*
 *      main.c
 */
#include "cabecera.h"
#include <stdio.h>

char    *Otra_cadena = "Hola a Todos";

main()
{
    printf("Ejecutando...\n");

    /*
     *      Llamar a EscribirMiCadena() - definida en otro archivo
     */
    EscribirMiCadena(MI_CADENA);

    printf("Terminado.\n");
}
```

## *EscribirMiCadena.c*

```
/*
 *      EscribirMiCadena.c
 */
extern char    *Otra_cadena;

void EscribirMiCadena(EstaCadena)
char    *EstaCadena;
{
    printf("%s\n", EstaCadena);
}
```

```
        printf("Variable Global = %s\n", Otra_cadena);  
    }
```

### ***cabecera.h***

```
/*  
 *      cabecera.h  
 */  
#define MI_CADENA "Hola Mundo"  
  
void EscribirMiCadena();
```

Cada módulo será compilado separadamente como se verá más adelante.

Algunos módulos tienen la directiva de preprocesamiento `#include "cabecera.h"` ya que comparten definiciones comunes. Algunos como *main.c* también incluyen archivos cabecera estándar. La función *main.c* llama a la función `EscribirMiCadena()` la cual esta en el módulo (archivo) *EscribirMiCadena.c*

El prototipo `void` de la función `EscribirMiCadena` esta definida en *cabecera.h*.

Observar que en general se debe decidir entre tener un módulo `.c` que tenga acceso solamente a la información que necesita para su trabajo, con la consecuencia de mantener muchos archivos cabecera y tener programas de tamaño moderado con uno o dos archivos cabecera (probablemente lo mejor) que compartan más definiciones de módulos.

Un problema que se tiene al emplear módulos son el compartir variables. Si se tienen variables globales declaradas y son instanciadas en un módulo, ¿cómo pueden ser pasadas a otros módulos para que sean conocidas?

Se podrían pasar los valores como parámetros a las funciones, pero:

- puede ser esto laborioso si se pasan los mismos parámetros a muchas funciones o si la lista de argumentos es muy larga.
- arreglos y estructuras muy grandes son difíciles de guardar localmente -- problemas de memoria con el stack.

## **21.2 Variables y Funciones Externas**

Las variables y argumentos definidos dentro de las funciones son "internas", es decir, locales.

Las variables "externas" están definidas fuera de las funciones -- se encuentran potencialmente disponibles a todo el programa (globales) pero NO necesariamente. Las variables externas son siempre permanentes.

En el lenguaje C, todas las definiciones de funciones son externas, NO se pueden tener declaraciones de funciones anidadas como en PASCAL.

## 21.2.1 Alcance de las variables externas

Una variable externa (o función) no es siempre totalmente global. En el lenguaje C se aplica la siguiente regla:

*El alcance de una variable externa (o función) inicia en el punto de declaración hasta el fin del archivo (módulo) donde fue declarada.*

Considerar el siguiente código:

```
main()
{ ... }

int que_alcance;
float fin_de_alcance[10];

void que_global()
{ ... }

char solitaria;

float fn()
{ ... }
```

La función `main()` no puede ver a las variables `que_alcance` o `fin_de_alcance`, pero las funciones `que_global()` y `fn()` si pueden. Solamente la función `fn()` puede ver a `solitaria`.

Esta es también una de las razones por las que se deben poner los *prototipos* de las funciones antes del cuerpo del código.

Por lo que en el ejemplo la función `main` no conocerá nada acerca de las funciones `que_global()` y `fn()`. La función `que_global()` no sabe nada acerca de la función `fn()`, pero `fn()` si sabe acerca de la función `que_global()`, ya que esta aparece declarada previamente.

*La otra razón por la cual se usan los prototipos de las funciones es para revisar los parámetros que serán pasados a las funciones.*

Si se requiere hacer referencia a una variable externa antes de que sea declarada o que esta definida en otro módulo, la variable debe ser declarada como una variable externa, por ejemplo:

```
extern int que_global;
```

Regresando al ejemplo de programación modular, se tiene una arreglo de caracteres tipo global `Otra_cadena` declarado en *main.c* y que esta compartido con *EscribirMiCadena* donde esta declarada como externa.

Se debe tener cuidado con el *especificador de almacenamiento de clase* ya que el prefijo es una *declaración*, NO una *definición*, esto es, no se da almacenamiento en la memoria para una variable externa -- solamente le dice al compilador la propiedad de la variable. La variable actual sólo deberá estar definida una vez en todo el programa -- se pueden tener tantas declaraciones externas como se requieran.

Los tamaños de los arreglos deberán ser dados dentro de la declaración, pero no son necesarios en las declaraciones externas, por ejemplo:

```
main.c      int arr[100];
arch.c     extern int arr[];
```

## 21.3 Ventajas de Usar Varios Archivos

Las ventajas principales de dispersar un programa en varios archivos son:

- Equipos de programadores pueden trabajar en el programa, cada programador trabaja en un archivo diferente.
- Puede ser usado un estilo orientado a objetos. Cada archivo define un tipo particular de objeto como un tipo de dato y las operaciones en ese objeto como funciones. La implementación del objeto puede mantenerse privado al resto del programa. Con lo anterior se logran programas bien estructurados los cuales son fáciles de mantener.
- Los archivos pueden contener todas las funciones de un grupo relacionado, por ejemplo todas las operaciones con matrices. Estas pueden ser accesadas como una función de una biblioteca.
- Objetos bien implementados o definiciones de funciones pueden ser reusadas en otros programas, con lo que se reduce el tiempo de desarrollo.
- En programas muy grandes cada función principal puede ocupar un propio archivo. Cualquier otra función de bajo nivel usada en la implementación puede ser guardada en el mismo archivo, por lo que los programadores que llamen a la función principal no se distraerán por el trabajo de bajo nivel.
- Cuando los cambios son hechos a un archivo, solamente ese archivo necesita ser recompilado para reconstruir el programa. La utilidad `make` de UNIX es muy útil para reconstruir programas con varios archivos.

## 21.4 Como dividir un programa en varios archivos

Cuando un programa es separado en varios archivos, cada archivo contendrá una o más funciones. Un archivo incluirá la función `main()` mientras los otros contendrán funciones que serán llamados por otros. Estos otros archivos pueden ser tratados como funciones de una biblioteca.

Los programadores usualmente inician diseñando un programa dividiendo el problema en secciones más fácilmente manejables. Cada una de estas secciones podrán ser implementadas como una o más funciones. Todas las funciones de cada sección por lo general estarán en un sólo archivo.

Cuando se hace una implementación tipo objeto de las estructuras de datos, es usual tener todas las funciones que accesan ése objeto en el mismo archivo. Las ventajas de lo anterior son:

- El objeto puede ser fácilmente reusado en otros programas.

- Todas las funciones relacionadas están guardadas juntas.
- Los últimos cambios al objeto requieren solamente la modificación de un archivo.

El archivo contiene la definición de un objeto, o funciones que regresan valores, hay una restricción en la llamada de estas funciones desde otro archivo, al menos que la definición de las funciones estén en el archivo, no será posible compilar correctamente.

La mejor solución a este problema es escribir un archivo cabecera para cada archivo de C, estos tendrán el mismo nombre que el archivo de C, pero terminarán en `.h`. El archivo cabecera contiene las definiciones de todas las funciones usadas en el archivo de C.

Cuando una función en otro archivo llame una función de nuestro archivo de C, se puede definir la función usando la directiva `#include` con el archivo apropiado `.h`

## 21.5 Organización de los Datos en cada Archivo

Cualquier archivo deberá tener sus datos organizados en un cierto orden, típicamente podrá ser la siguiente:

- Un preámbulo consistente de las definiciones de constantes (`#define`), cabeceras de archivos (`#include`) y los tipos de datos importantes (`typedef`).
- Declaración de variables globales y externas. Las variables globales podrían estar inicializadas aquí.
- Una o más funciones.

El orden anterior es importante ya que cada objeto deberá estar definido antes de que pueda ser usado. Las funciones que regresan valores deberán estar definidos antes de que sean llamados. Esta definición podría ser una de las siguientes:

- El lugar en que la función está definida y llamada en el mismo archivo, una declaración completa de la función puede ser colocada delante de cualquier llamada de la función.
- Si la función es llamada desde un archivo donde no está definida, un prototipo deberá aparecer antes que llamada de la función.

Una función definida como:

```
float enc_max(float a, float b, float c)
{ ... }
```

podrá tener el siguiente prototipo:

```
float enc_max(float a, float b, float c);
```

El prototipo puede aparecer entre las variables globales en el inicio del archivo fuente. Alternativamente puede ser declarado en el archivo cabecera el cual es leído usando la directiva `#include`.

Es importante recordar que todos los objetos en C deberán estar declarados antes de ser usados.

## 21.6 La utilería Make

La utilería *Make* es un manejador inteligente de programas que mantiene la integridad de una colección de módulos de un programa, una colección de programas o un sistema completo -- no tienen que ser programas, en la práctica puede ser cualquier sistema de archivos (por ejemplo, capítulos de texto de un libro que esta siendo tipografiado). Su uso principal ha sido en la asistencia del desarrollo de sistemas de software.

Esta utilería fue inicialmente desarrollada para UNIX, pero actualmente esta disponible en muchos sistemas.

Observar que `make` es una herramienta del programador, y no es parte del lenguaje C o de alguno otro.

Spongamos el siguiente problema de mantenimiento de una colección grande de archivos fuente:

```
main.c fl.c ..... fn.c
```

Normalmente se compilarán los archivos de la siguiente manera:

```
gcc -o main main.c fl.c ..... fn.c
```

Sin embargo si se sabe que algunos archivos han sido compilados previamente y sus archivos fuente no han sido cambiados desde entonces, entonces se puede ahorrar tiempo de compilación ligando los códigos objetos de estos archivos, es decir:

```
gcc -o main main.c fl.c ... fi.o ... fj.o ... fn.c
```

Se puede usar la opción `-c` del compilador de C para crear un código objeto (`.o`) para un módulo dado. Por ejemplo:

```
gcc -c main.c
```

que creará un archivo `main.o`. No se requiere proporcionar ninguna liga de alguna biblioteca ya que será resuelta en la etapa de ligamiento.

Se tiene el problema en la compilación del programa de ser muy larga, sin embargo:

- Se consume tiempo al compilar un módulo `.c` -- si el módulo ha sido compilado antes y no ha sido modificado el archivo fuente, por lo tanto no hay necesidad de recompilarlo. Se puede solamente ligar los archivos objeto. Sin embargo, no será fácil recordar cuales archivos han sido actualizados, por lo que si ligamos un archivo objeto no actualizado, el programa ejecutable final estará incorrecto.

- Cuando se teclea una secuencia larga de compilación en la línea de comandos se cometen errores de tecleo, o bien, se teclea en forma incompleta. Existirán varios de nuestros archivos que serán ligados al igual que archivos de bibliotecas del sistema. Puede ser difícil recordar la secuencia correcta.

Si se usa la utilidad `make` todo este control es hecho con cuidado. En general sólo los módulos que sean más viejos que los archivos fuente serán recompilados.

## 21.6.1 Programando Make

La programación de `make` es directa, básicamente se escribe una secuencia de comandos que describe como nuestro programa (o sistema de programas) será construido a partir de los archivos fuentes.

La secuencia de construcción es descrita en los archivos `makefile`, los cuales contienen *reglas de dependencia y reglas de construcción*.

Una regla de dependencia tiene dos partes -- un lado izquierdo y un lado derecho separados por :

```
lado izquierdo : lado derecho
```

El lado izquierdo da el nombre del destino (los nombres del programa o archivos del sistema) que será construido (*target*), mientras el lado derecho da los nombres de los archivos de los cuales depende el destino (por ejemplo, archivos fuente, archivos cabecera, archivos de datos).

Si el destino está fuera de fecha con respecto a las partes que le constituyen, las reglas de construcción siguiendo las reglas de dependencia son usadas.

Por lo tanto para un programa típico de C cuando un archivo `make` es ejecutado las siguientes tareas son hechas:

1. El archivo `make` es leído. El `Makefile` indica cuales objetos y archivos de biblioteca se requieren para ser ligados y cuales archivos cabecera y fuente necesitan ser compilados para crear cada archivo objeto.
2. La hora y la fecha de cada archivo objeto son revisados contra el código fuente y los archivos cabecera de los cuales dependen. Si cualquier fuente o archivo cabecera son más recientes que el archivo objeto, entonces los archivos han sido modificados desde la última modificación y por lo tanto los archivos objeto necesitan ser recompilados.
3. Una vez que todos los archivos objetos han sido revisados, el tiempo y la fecha de todos los archivos objeto son revisados contra los archivos ejecutables. Si existe archivos ejecutables viejos serán recompilados.

Observar que los archivos `make` pueden obedecer cualquier comando que sea tecleado en la línea de comandos, por consiguiente se pueden usar los archivos `make` para no solamente compilar archivos, sino también para hacer respaldos, ejecutar programas si los archivos de datos han sido cambiados o limpieza de directorios.

## 21.7 Creación de un Archivo Make (Makefile)

La creación del archivo es bastante simple, se crea un archivo de texto usando algún editor de textos. El archivo *Makefile* contiene solamente una lista de dependencias de archivos y comandos que son necesarios para satisfacerlos.

Se muestra a continuación un ejemplo de un archivo make:

```
prog: prog.o f1.o f2.o
    gcc -o prog prog.o f1.o f2.o -lm ...

prog.o: cabecera.h prog.c
    gcc -c prog.c

f1.o: cabecera.h f1.c
    gcc -c f1.c

f2.o: ....
    ...
```

La utilidad `make` lo interpretará de la siguiente forma:

1. `prog` depende de tres archivos: `prog.o`, `f1.o` y `f2.o`. Si cualquiera de los archivos objeto ha cambiado desde la última compilación los archivos deben ser religados.
2. `prog.o` depende de 2 archivos, si estos han cambiado `prog.o` deberá ser recompilado. Lo mismo sucede con `f1.o` y `f2.o`.

Los últimos 3 comandos en `makefile` son llamados *reglas explícitas* -- ya que los archivos en los comandos son listados por nombre.

Se pueden usar *reglas implícitas* en `makefile` para generalizar reglas y hacer más compacta la escritura.

Si se tiene:

```
f1.o: f1.c
    gcc -c f1.c

f2.o: f2.c
    gcc -c f2.c
```

se puede generalizar a:

```
.c.o: gcc -c $<
```

Lo cual se lee como `.ext_fuente.ext_destino: comando` donde `$<` es una forma breve para indicar los archivos que tienen la extensión `.c`

Se pueden insertar comentarios en un `Makefile` usando el símbolo `#`, en donde todos los caracteres que siguen a `#` son ignorados.

## 21.8 Uso de macros con Make

Se pueden definir *macros* para que sean usadas por `make`, las cuales son típicamente usadas para guardar nombres de archivos fuente, nombres de archivos objeto, opciones del compilador o ligas de bibliotecas.

Se definen en una forma simple, por ejemplo:

```
FUENTES          = main.c f1.c f2.c
CFLAGS           = -ggdb -C
LIBS             = -lm
PROGRAMA         = main
OBJETOS          = (FUENTES: .c = .o)
```

en donde `(FUENTES: .c = .o)` cambia la extensión `.c` de los fuentes por la extensión `.o`

Para referirse o usar una macro con `make` se debe hacer `$(nomb_macro)`, por ejemplo:

```
$(PROGRAMA) : $(OBJETOS)
$(LINK.C) -o $@ $(OBJETOS) $(LIBS)
```

En el ejemplo mostrado se observa que:

- La línea que contiene `$(PROGRAMA) : $(OBJETOS)` genera una lista de dependencias y el destino.
- Se emplean macros internas como `$@`.

Existen varias macros internas a continuación se muestran algunas de ellas:

```
$*
    Parte del nombre del archivo de la dependencia actual sin el sufijo.
$@
    Nombre completo del destino actual.
$
    Archivo .c del destino
```

Un ejemplo de un `makefile` para el programa modular discutido previamente se muestra a continuación:

```
#
# Makefile
#
FUENTES.c=main.c EscribirMiCadena.c
INCLUDES=
CFLAGS=
SLIBS=
PROGRAMA=main

OBJETOS=$(FUENTES.c:.c=.o)

# Destino (target) especial (inicia con .)
.KEEP_STATE:

debug := CFLAGS=-ggdb
```

```
all debug: $(PROGRAMA)

$(PROGRAMA): $(INCLUDES) $(OBJETOS)
             $(LINK.c) -o $@ $(OBJETOS) $(SLIBS)

clean:
    rm -f $(PROGRAMA) $(OBJETOS)
```

Para ver más información de esta utilidad dentro de *Linux* usar `info make`

## 21.9 Ejecución de Make

Para usar `make` solamente se deberá teclear en la línea de comandos. El sistema operativo automáticamente busca un archivo con el nombre `Makefile` (observar que la primera letra es mayúscula y el resto minúsculas), por lo tanto si se tiene un archivo con el nombre `Makefile` y se teclea `make` en la línea de comandos, el archivo `Makefile` del directorio actual será ejecutado.

Se puede anular esta búsqueda de este archivo tecleando `make -f makefile`.

Existen algunas otras opciones para `make` las cuales pueden ser consultadas usando `man`.

---

---

## Subsecciones

- [22.1 Entubando en un programa de C <stdio.h>](#)
    - [22.1.1 popen\(\) Tubería formateada](#)
    - [22.1.2 pipe\(\) Tubería de bajo nivel](#)
- 

# 22. Comunicación entre procesos (IPC Interprocess Communication), PIPES

A continuación se iniciará la revisión de como múltiples procesos pueden estar siendo ejecutados en una máquina y quizás siendo controlados (generados por la función `fork()`) por uno de nuestros programas.

En numerosas aplicaciones hay una necesidad clara para que estos procesos se comuniquen para el intercambio de datos o información de control. Hay unos cuantos métodos para hacer lo anterior. Se pueden considerar los siguientes:

- Tubería (Pipes)
- Señales (Signals)
- Colas de Mensajes
- Semáforos
- Memoria Compartida
- Sockets

En este capítulo se estudia el entubamiento de dos procesos, en los siguientes capítulos se revisan los otros métodos.

## 22.1 Entubando en un programa de C <stdio.h>

El entubamiento es un proceso donde la entrada de un proceso es hecha con la salida de otro. Se ha visto ejemplos de lo anterior en la línea de comandos de Unix cuando se usa el símbolo `|`.

Se verá como hacer lo anterior en un programa de C teniendo dos (o más) procesos divididos. Lo primero que se debe hacer es abrir una tubería, en Unix permite dos formas de hacerlo:

### 22.1.1 popen() Tubería formateada

FILE \*popen(char \*orden, char \*tipo) Abre una tubería de E/S donde el comando es

el proceso que será conectado al proceso que lo está llamando de esta forma creando el *pipe*. El tipo es "r" -- para lectura, o "w" para escritura.

La función `popen()` regresa un apuntador a un flujo o NULL para algún tipo de error.

Una tubería abierta con `popen()` deberá siempre cerrarse con `pclose(FILE *flujo)`.

Se usa `fprintf()` y `fscanf()` para comunicarse con el flujo de la tubería.

## 22.1.2 `pipe()` Tubería de bajo nivel

`int pipe(int descf[2])` Crea un par de descriptores de archivo, que apuntan a un *nodo* *i* de una tubería, y los pone en el vector de dos elementos apuntado por `descf`. `descf[0]` es para lectura y `descf[1]` es para escritura.

`pipe()` regresa 0 en caso de éxito, y en caso de error se devuelve -1 y se pone un valor apropiado en `errno`.

El modelo de programación estándar es que una vez que la tubería ha sido puesta, dos (o más) procesos cooperativos serán creados por división y los datos serán pasados empleando `read()` y `write()`.

Las tuberías abiertas con `pipe()` deberán ser cerradas con `close(int fd)`. A continuación se muestra un ejemplo de entubamiento que se utiliza para estar enviando datos al programa `gnuplot`, empleado para graficar. La descripción de las funciones de los módulos es la siguiente:

- El programa tiene dos módulos, `grafica.c` que contiene la función `main()` y `graficador.c` que contiene las rutinas para enviar los datos al programa `gnuplot`.
- El programa supone que se tiene instalado el programa de graficación `gnuplot` en el directorio `/usr/bin`.
- El programa `grafica.c` llama al programa `gnuplot`.
- Dos flujos son generados usando el programa `grafica.c`, en uno de ellos se gráfica la función  $y=0.5$  e  $y=\text{aleat}(0-1.0)$  y en el otro las funciones  $y=\text{sen}(x)$  e  $y=\text{sen}(1/x)$ .
- Dos tuberías son creadas cada una con un flujo de datos.
- `Gnuplot` genera en ``vivo" la salida de las gráficas.

El listado del código para `grafica.c` es el siguiente:

```
/* grafica.c -
 * Ejemplo de la tuberías (pipe) de Unix. Se llama al paquete "gnuplot"
 * para graficar desde un programa en C.
 * La información es entubada a gnuplot, se crean 2 tuberías, una dibujara la
 * grafica de y=0.5 e y= random 0-1.0, la otra graficara y=sen(1/x) e y=sen x
 */

#include "externals.h"
#include <signal.h>

#define DEG_TO_RAD(x) (x*180/M_PI)

void salir();
```

```

FILE *fp1, *fp2, *fp3, *fp4;

main()
{
    float i;
    float y1,y2,y3,y4;

    /* abrir archivos en los cuales se guardaran los datos a graficar */
    if ( ((fp1 = fopen("plot11.dat","w")) == NULL) ||
        ((fp2 = fopen("plot12.dat","w")) == NULL) ||
        ((fp3 = fopen("plot21.dat","w")) == NULL) ||
        ((fp4 = fopen("plot22.dat","w")) == NULL) )
        { printf("Error no se puede abrir uno o varios archivos de
datos\n");
        exit(1);
        }

    signal(SIGINT,salir); /* Atrapar la llamada de la funcion de salida ctrl-c
*/
    IniciarGraf();
    y1 = 0.5;
    srand48(1); /* poner semilla */
    for (i=0;;i+=0.01) /* incrementar i siempre */
    {
        /* usar ctrl-c para salir del programa */
        y2 = (float) drand48();
        if (i == 0.0)
            y3 = 0.0;
        else
            y3 = sin(DEG_TO_RAD(1.0/i));
        y4 = sin(DEG_TO_RAD(i));

        /* cargar archivos */
        fprintf(fp1,"%f %f\n",i,y1);
        fprintf(fp2,"%f %f\n",i,y2);
        fprintf(fp3,"%f %f\n",i,y3);
        fprintf(fp4,"%f %f\n",i,y4);

        /* asegurarse que los buffers son copiados al disco */
        /* para que gnuplot pueda leer los datos */
        fflush(fp1);
        fflush(fp2);
        fflush(fp3);
        fflush(fp4);

        /* graficar alot graph */
        Graf1Vez();
        usleep(10000); /* dormir por un corto tiempo (250 microsegundos) */
    }
}

void salir()
{
    printf("\nctrl-c atrapada:\n Terminando las tuberias\n");
    PararGrafica();

    printf("Cerrando los archivos de datos\n");
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    fclose(fp4);

    printf("Borrando los archivos de datos\n");
    BorrarDat();
}

```

El módulo graficador .c es el siguiente:

```
/* **** */
/* modulo: graficador.c */
/* Contiene rutinas para graficar un archivo de datos producido por */
/* programa. En este caso se grafica en dos dimensiones */
/* **** */

#include "externals.h"

static FILE *plot1,
           *plot2,
           *ashell;

static char *iniciargraf1 = "plot [] [0:1.1] 'plot11.dat' with lines,
'plot12.dat' with lines\n";

static char *iniciargraf2 = "plot 'plot21.dat' with lines, 'plot22.dat' with
lines\n";

static char *comando1= "/usr/bin/gnuplot> dump1";
static char *comando2= "/usr/bin/gnuplot> dump2";
static char *borrarchivos = "rm plot11.dat plot12.dat plot21.dat plot22.dat";
static char *set_term = "set terminal x11\n";

void IniciarGraf(void)
{
    plot1 = popen(comando1, "w");
    fprintf(plot1, "%s", set_term);
    fflush(plot1);
    if (plot1 == NULL)
        exit(2);
    plot2 = popen(comando2, "w");
    fprintf(plot2, "%s", set_term);
    fflush(plot2);
    if (plot2 == NULL)
        exit(2);
}

void BorrarDat(void)
{
    ashell = popen(borrarchivos, "w");
    exit(0);
}

void PararGrafica(void)
{
    pclose(plot1);
    pclose(plot2);
}

void Graf1Vez(void)
{
    fprintf(plot1, "%s", iniciargraf1);
    fflush(plot1);

    fprintf(plot2, "%s", iniciargraf2);
    fflush(plot2);
}
```

El archivo de cabecera externals.h contiene lo siguiente:

```
/* externals.h */
#ifndef EXTERNALS
#define EXTERNALS
```

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

/* prototipos */
```

Con la finalidad de tener un mejor control en la compilación del programa se puede usar el siguiente archivo Makefile:

```
FUENTES.c= grafica.c graficador.c
INCLUDES=
CFLAGS=
SLIBS= -lm
PROGRAMA= grafica

OBJETOS= $(FUENTES.c:.c=.o)

$(PROGRAMA): $(INCLUDES) $(OBJETOS)
    gcc -o $(PROGRAMA) $(OBJETOS) $(SLIBS)

grafica.o: externals.h grafica.c
    gcc -c grafica.c

graficador.o: externals.h graficador.c
    gcc -c graficador.c

clean:
    rm -f $(PROGRAMA) $(OBJETOS)
```

---

---

## Subsecciones

- [23.1 Creación y nombrado de sockets](#)
  - [23.2 Conectando sockets de flujo](#)
    - [23.2.1 Transferencia de datos en un flujo y cerrado](#)
- 

# 23. Sockets

Los sockets proporcionan una comunicación de dos vías, punto a punto entre dos procesos. Los sockets son muy versátiles y son un componente básico de comunicación entre interprocesos e intersistemas. Un socket es un punto final de comunicación al cual se puede asociar un nombre. Este tiene un tipo y uno o más procesos asociados.

Los sockets existen en los dominios de comunicación. Un socket de dominio es una representación que da una estructura de direccionamiento y un conjunto de protocolos. Los sockets se conectan solamente con sockets en el mismo dominio. Veintitrés dominios de sockets son identificados (ver `<sys/socket.h>`), de los cuales solamente los dominios de UNIX e Internet son normalmente sockets de Linux usados para comunicarse entre procesos en un sólo sistema, como otras formas de comunicación entre procesos.

El dominio de UNIX da un espacio de direcciones de socket en un sistema. Los sockets de dominio de UNIX son nombrados con las rutas de UNIX. Los sockets pueden también ser usados para comunicar procesos entre diferentes sistemas. El espacio de direcciones del socket entre los sistemas conectados es llamado el dominio de Internet.

La comunicación del dominio de Internet usa la suite del protocolo de Internet TCP/IP.

Los *tipos de socket* definen las propiedades de comunicación visibles para la aplicación. Los procesos se comunican solamente entre los sockets del mismo tipo. Existen cinco tipos de sockets.

### Socket de flujo

da un flujo de datos de dos vías, confiable, y sin duplicados sin límites de grabación. El flujo opera en forma parecida a una conversación telefónica. El tipo del socket es *SOCK\_STREAM*, el cual en el dominio de Internet usa TCP (Transmission Control Protocol).

### Socket de datagrama

soporta un flujo de mensajes de dos vías. En un socket de datagrama podría recibir mensajes en diferente orden de la secuencia de la cual los mensajes fueron enviados. Los límites de grabación en los datos son preservados. Los sockets de datagrama operan parecidos a pasar cartas hacia adelante y hacia atrás en el correo. El tipo de socket es *SOCK\_DGRAM*, el cual en el dominio de internet usa UDP (User Datagram Protocol).

### Socket de paquete secuencial

da una conexión de dos vías, secuencial y confiable para datagramas de una longitud fija

máxima. El tipo de socket es *SOCK\_SEQPACKET*. No hay protocolo implementado para este tipo de cualquier familia de protocolos.

### raw socket

da acceso a los protocolos de comunicación subyacente.

Los sockets son usualmente datagramas orientados, pero sus características exactas dependen de la interfaz dada por el protocolo.

## 23.1 Creación y nombrado de sockets

Se llama a la función `int socket(int dominio, int tipo, int protocolo);` para crear un extremo de una comunicación (socket) en el dominio especificado y del tipo indicado. Si un protocolo no es especificado, el sistema usa uno predefinido que soporte el tipo de socket especificado. El manejador del socket (un descriptor) es devuelto. Un proceso remoto no tiene forma de identificar un socket hasta que una dirección se ligada a este. Los procesos de comunicación se conectan a través de direcciones. En el dominio de UNIX, una conexión esta compuesta usualmente de uno o dos nombres de rutas. En el dominio de Internet, una conexión esta compuesta de direcciones locales y remotas y partes locales y remotas. En muchos dominios, las conexiones deben ser únicas.

Se llama a la función `int bind(int sockfd, struct sockaddr *nomb, socklen_t longdir)` para enlazar un camino o una dirección de Interne a un conector (socket). Hay tres formas diferentes de llamar a `bind()`, dependiendo del dominio del socket.

- Para los sockets del dominio de UNIX con rutas conteniendo 14, o menos caracteres se puede usar:

```
#include <sys/socket.h>
...
bind (sd, (struct sockaddr *) &direccion, longitud);
```

- Si la trayectoria de un socket del dominio de UNIX requieres más caracteres, usar:

```
#include <sys/un.h>
...
bind (sd, (struct sockaddr_un *) &direccion, longitud);
```

- Para sockets del dominio de Internet usar:

```
#include <net/netinet.h>
...
bind (sd, (struct sockaddr_in *) &direccion, longitud);
```

En el dominio de UNIX, el enlazado de nombres crea un socket con nombre en el sistema de archivos. Usar `unlink()` or `rm()` para eliminar el socket.

## 23.2 Conectando sockets de flujo

La conexión de sockets es usualmente asimétrica. Un proceso usualmente actúa como un servidor y el otro proceso es el cliente. El servidor enlaza su socket a un camino o dirección previamente acordada, y entonces bloquea el socket. Para un conector `SOCK_STREAM`, el servidor llama a la función `int listen(int s, int backlog)`, para indicar cuantas peticiones de conexión serán puestas en la cola. Un cliente inicia la conexión al socket del servidor mediante una llamada a la función `int connect(int s, struct sockaddr *nomb_serv, int longdirec)`. Una llamada en el dominio de UNIX es como la siguiente:

```
struct sockaddr_un server;
...
connect ( sd, (struct sockaddr_un *)&server, long);
```

mientras en el dominio de Internet la llamada podría ser:

```
struct sockaddr_in server;
...
connect ( sd, (struct sockaddr_un *)&server, long);
```

Si el socket del cliente no esta enlazado al momento de hacer la llamada para la conexión, el sistema automáticamente selecciona y liga un nombre al socket. Para un socket `SOCK_STREAM`, el servidor llama a la función `accept()` para completar la conexión.

La función `int accept(int s, struct sockaddr *addr, int *longdirec)` regresa un conector nuevo el cual es válido solamente para la conexión particular. Un servidor puede tener múltiples conexiones `SOCK_STREAM` activas al mismo tiempo.

### 23.2.1 Transferencia de datos en un flujo y cerrado

Se emplean varias funciones para enviar y recibir datos de un socket `SOCK_STREAM`. Estas son `write()`, `read()`, `int send(int s, const void *msg, size_t lon, int flags)`, y `int recv(int s, void *buf, size_t lon, int flags)`. Las funciones `send()` y `recv()` son parecidas a `read()` y `write()`, pero tienen algunas banderas operacionales adicionales.

El argumento `flags` de una llamada a `recv` se forma aplicando el operador de bits O-lógico a uno o más de los valores siguientes:

#### **MSG\_OOB**

Pide la recepción de datos fuera-de-banda que no se reciban en el flujo de datos normal. Algunos protocolos ponen datos despachados con prontitud en la cabeza de la cola de datos normales, y así, esta opción no puede emplearse con tales protocolos.

#### **MSG\_PEEK**

Hace que la operacin de recepción devuelva datos del principio de la cola de recepción sin quitarlos de allí. Así, una próxima llamada de recepción devolverá los mismos datos.

#### **MSG\_WAITALL**

Esta opción hace que la operación se bloquee hasta que se satisfaga la petición completamente. Sin embargo, la llamada puede aun devolver menos datos de los pedidos si se captura una señal, si ocurre un error o una desconexión, o si los próximos datos que se van a recibir son de un tipo diferente del que se ha devuelto.

### **MSG\_NOSIGNAL**

Esta opción desactiva el que se produzca una señal SIGPIPE sobre los conectores orientados a conexión cuando el otro extremo desaparece.

### **MSG\_ERRQUEUE**

Esta opción indica que los errores encolados deben recibirse desde la cola de errores de conectores. El error se pasa en un mensaje auxiliar con un tipo dependiente del protocolo (para IPv4 este es IP\_RECVERR). El usuario debe proporcionar un buffer de tamaño suficiente. Vea `msg(3)` para obtener más información sobre mensajes auxiliares.

---

---

# Sobre este documento...

---

## Manual de C

This document was generated using the [LaTeX2HTML](#) translator Version 2002-2-1 (1.70)

Copyright © 1993, 1994, 1995, 1996, [Nikos Drakos](#), Computer Based Learning Unit, University of Leeds.

Copyright © 1997, 1998, 1999, [Ross Moore](#), Mathematics Department, Macquarie University, Sydney.

The command line arguments were:

**latex2html** manual

The translation was initiated by HÃ©ctor Tejeda V on 2005-08-12

---