

Capítulo 2. Escribiendo un buen software GNU / Linux



Este capítulo cubre algunas técnicas básicas que utilizan la mayoría de los programadores GNU / Linux. Siguiendo las pautas presentadas, podrá escribir programas que funcionen bien dentro del entorno GNU / Linux y cumplan con las expectativas de los usuarios de GNU / Linux sobre cómo deberían operar los programas.

2.1 Interacción con el entorno de ejecución

Cuando estudió C o C ++ por primera vez, aprendió que la especial [principal](#) La función es el punto de entrada principal para un programa. Cuando el sistema operativo ejecuta su programa, automáticamente proporciona ciertas funciones que ayudan al programa a comunicarse con el sistema operativo y el usuario. Probablemente haya aprendido acerca de los dos parámetros para [principal](#), generalmente llamado [argc](#) y [argv](#), que reciben entradas a su programa. Aprendiste sobre el [stdout](#) y [stdin](#) (o la [cout](#) y [cin](#) streams en C ++) que proporcionan entrada y salida de consola. Estas características las proporcionan los lenguajes C y C ++, e interactúan con el sistema GNU / Linux de ciertas formas. GNU / Linux también proporciona otras formas de interactuar con el entorno operativo.

2.1.1 La lista de argumentos

Ejecuta un programa desde un indicador de shell escribiendo el nombre del programa. Opcionalmente, puede proporcionar información adicional al programa escribiendo una o más palabras después del nombre del programa, separadas por espacios. Estos se llaman *argumentos de la línea de comandos*. (También puede incluir un argumento que contenga un espacio, encerrando el argumento entre comillas). De manera más general, esto se conoce como el programa *lista de argumentos* porque no es necesario que se origine en una línea de comandos de shell. En [Capítulo 3](#), "Procesos", verá otra forma de invocar un programa, en la que un programa puede especificar la lista de argumentos de otro programa directamente.

Cuando se invoca un programa desde el shell, la lista de argumentos contiene toda la línea de comandos, incluido el nombre del programa y cualquier argumento de la línea de comandos que se haya proporcionado. Suponga, por ejemplo, que invoca el `ls` comando en su shell para mostrar el contenido del directorio raíz y los tamaños de archivo correspondientes con esta línea de comando:

```
% ls -s /
```

La lista de argumentos que el `ls` El programa recibe tiene tres elementos. El primero es el nombre del programa en sí, como se especifica en la línea de comando, a saber `ls`. El segundo y tercer elemento de la lista de argumentos son los dos argumentos de la línea de comandos, `-s` y `/`.

los [principal](#) función de su programa puede acceder a la lista de argumentos a través de la [argc](#) y [argv](#) parámetros a [principal](#) (si no los usa, simplemente puede omitirlos). El primer parámetro, [argc](#), es un número entero que se establece en el número de elementos de la lista de argumentos. El segundo parámetro, [argv](#), es una matriz de

punteros de carácter. El tamaño de la matriz `esargc`, y los elementos de la matriz apuntan a los elementos de la lista de argumentos, como cadenas de caracteres terminadas en NUL.

Usar argumentos de línea de comandos es tan fácil como examinar el contenido de `argc` y `argv`. Si no está interesado en el nombre del programa en sí, no olvide omitir el primer elemento.

[Listado 2.1](#) demuestra cómo usar `argc` y `argv`.

Listado 2.1 (*arglist.c*) Usando `argc` y `argv`

incluye <stdio.h>

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

    printf ("El nombre de este programa es '%s'. \n", argv [0]);
    printf ("Este programa fue invocado con %d argumentos. \n", argc - 1);

    /* ¿Se especificó algún argumento en la línea de comandos? */ if
    (argc > 1) {
        /* Sí, imprímalos. */ int i;

        printf ("Los argumentos son: \n"); para (i =
            1; i < argc; ++ i) printf ("%s \n", argv [i]);

    }

    return 0;
}
```

2.1.2 Convenciones de la línea de comandos de GNU / Linux

Casi todos los programas GNU / Linux obedecen algunas convenciones sobre cómo se interpretan los argumentos de la línea de comandos. Los argumentos que esperan los programas se dividen en dos categorías: *opciones* (o *banderas*) y otros argumentos. Las opciones modifican cómo se comporta el programa, mientras que otros argumentos proporcionan entradas (por ejemplo, los nombres de los archivos de entrada).

Las opciones vienen en dos formas:

- *Opciones cortas* constan de un solo guión y un solo carácter (generalmente una letra minúscula o mayúscula). Las opciones cortas son más rápidas de escribir.
- *Opciones largas* constan de dos guiones, seguidos de un nombre formado por letras minúsculas y mayúsculas y guiones. Las opciones largas son más fáciles de recordar y de leer (en scripts de shell, por ejemplo).

Por lo general, un programa proporciona tanto una forma corta como una forma larga para la mayoría de las opciones que admite, la primera por brevedad y la segunda por claridad. Por ejemplo, la mayoría de los programas comprenden las opciones `-h` y `-ayuda`, y tratarlos de manera idéntica. Normalmente, cuando se invoca un programa desde el shell, cualquier opción deseada sigue inmediatamente al nombre del programa. Algunas opciones esperan un argumento inmediatamente después. Muchos programas, por ejemplo, interpretan la opción `-salida foo` para especificar que la salida del programa debe colocarse en un archivo llamado `foo`. Después de las opciones, pueden seguir otros argumentos de la línea de comandos, normalmente archivos de entrada o datos de entrada.

Por ejemplo, el comando `ls -s /` muestra el contenido del directorio raíz. `ls -s` La opción modifica el comportamiento predeterminado de `ls` indicándole que muestre el tamaño (en kilobytes) de cada entrada. `ls /` el argumento dice `ls` qué directorio listar. `ls -Talla` opción es sinónimo de `-s`, por lo que se podría haber invocado el mismo comando como `ls --tamaño /`.

los *Estándares de codificación GNU* enumere los nombres de algunas opciones de línea de comandos de uso común. Si planea proporcionar opciones similares a estas, es una buena idea usar los nombres especificados en los estándares de codificación. Su programa se comportará más como otros programas y será más fácil de aprender para los usuarios. Puede ver las pautas de los Estándares de codificación GNU para las opciones de la línea de comandos invocando lo siguiente desde un indicador de shell en la mayoría de los sistemas GNU / Linux:

% info "(estándares) Interfaces de usuario"

2.1.3 Uso *getopt_long*

Analizar las opciones de la línea de comandos es una tarea tediosa. Afortunadamente, la biblioteca GNU C proporciona una función que puede usar en programas C y C ++ para hacer este trabajo algo más fácil (aunque todavía un poco molesto). Esta función, *getopt_long*, entiende tanto las opciones cortas como las largas. Si usa esta función, incluya el archivo de encabezado `<getopt.h>`.

Suponga, por ejemplo, que está escribiendo un programa que debe aceptar las tres opciones que se muestran en [Cuadro 2.1](#).

Cuadro 2.1. Opciones de programa de ejemplo

Forma corta	Forma larga	Objetivo
- h	-- ayuda	Mostrar resumen de uso y salir
- o <i>nombre del archivo</i>	-- producción <i>nombre del archivo</i>	Especificar el nombre del archivo de salida
- v	-- detallado	Imprimir mensajes detallados

Además, el programa debe aceptar cero o más argumentos de línea de comandos adicionales, que son los nombres de los archivos de entrada.

Usar *getopt_long*, debe proporcionar dos estructuras de datos. La primera es una cadena de caracteres que contiene las opciones breves válidas, cada una con una sola letra. Una opción que requiere un argumento va seguida de dos puntos. Para su programa, la cadena `ho:v` indica que las opciones válidas son `-h`, `-o`, y `-v`, con la segunda de estas opciones seguida de un argumento.

Para especificar las opciones largas disponibles, construya una matriz de *opción de estructura* elementos. Cada elemento corresponde a una opción larga y tiene cuatro campos. En circunstancias normales, el primer campo es el nombre de la opción larga (como una cadena de caracteres, sin los dos guiones); el segundo es 1 si la opción tiene un argumento, o 0 en caso contrario; el tercero es `NULL`; y el cuarto es una constante de carácter que especifica el sinónimo de opción corta para esa opción larga. El último elemento de la matriz debe ser todo ceros. Podría construir la matriz de esta manera:

```
opción de estructura const long_options [] = {
    {"ayuda", 0, NULL, 'h'},
    {"salida", 1, NULL, 'o'},
    {}
};
```

```

{"verbose", 0, NULL, 'v'}, {NULL, 0,
NULL, 0}
};

```

Invoca el `getopt_long` función, pasándole el `argc` y `argv` argumentos para `principal`, la cadena de caracteres que describe opciones breves, y la matriz de `opción de estructura` elementos que describen las opciones largas.

- Cada vez que llamas `getopt_long`, analiza una sola opción, devolviendo la letra de opción corta para esa opción, o -1 si no se encuentran más opciones.
- Normalmente, llamarás `getopt_long` en un bucle, para procesar todas las opciones que el usuario ha especificado, y manejará las opciones específicas en una declaración de cambio.
- Si `getopt_long` encuentra una opción no válida (una opción que no especificó como una opción válida corta o larga), imprime un mensaje de error y devuelve el carácter `?` (un signo de interrogación). La mayoría de los programas se cerrarán en respuesta a esto, posiblemente después de mostrar información de uso.
- Al manejar una opción que toma un argumento, la variable global `optarg` apunta al texto de ese argumento.
- Después `getopt_long` ha terminado de analizar todas las opciones, la variable global `optind` contiene el índice (en `argv`) del primer argumento de no opción.

[Listado 2.2](#) muestra un ejemplo de cómo puede utilizar `getopt_long` para procesar sus argumentos.

Listado 2.2 (getopt_long.c) Usando getopt_long

```

# incluye <getopt.h>
# incluye <stdio.h>
# incluye <stdlib.h>

/* El nombre de este programa. */ const

char * nombre_programa;

/* Imprime la información de uso de este programa en STREAM (normalmente
   stdout o stderr) y salga del programa con EXIT_CODE. No vuelve. */

void print_usage (ARCHIVO * flujo, int código_salida) {

    fprintf (flujo, "Uso:% s opciones [archivo de entrada ....] \ n", nombre_programa); fprintf (corriente,

        "-h --help Muestra esta información de uso. \ n"
        "-o --output filename Escribe la salida en el archivo. \ n" "-v --verbose
        Imprime mensajes detallados. \ n");
    salir (código_salida);
}

/* Punto de entrada del programa principal. ARGV contiene el número de lista de argumentos
   elementos; ARGV es una serie de indicadores para ellos. */

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

    int next_option;

    /* Una cadena que enumera letras de opciones cortas válidas. */
    const char * const short_options = "ho: v";

```

```

/* Una matriz que describe opciones largas válidas. */
const struct option long_options [] = {"help", 0, NULL, 'h'},

{"salida", 1, NULL, 'o'}, {"verbose", 0,
NULL, 'v'}, {NULL, 0, NULL, 0}

/* Requerido al final de la matriz. */

};

/* El nombre del archivo para recibir la salida del programa, o NULL para
salida estándar. */
const char * nombre_archivo_salida = NULL;
/* Si mostrar mensajes detallados. */
int verbose = 0;

/* Recuerda el nombre del programa, para incorporarlo en los mensajes.
El nombre se almacena en argv [0]. */
nombre_programa = argv [0];

hacer {
    next_option = getopt_long (argc, argv, short_options,
                                opciones_largas, NULL);

    cambiar (siguiente_opción)
    {
        caso 'h': /* -h o --help */
            /* El usuario ha solicitado información de uso. Imprímelo a estándar
            salida y salir con el código de salida cero (terminación normal). */ print_usage
            (salida estándar, 0);

        caso 'o': /* -o o --salida */
            /* Esta opción toma un argumento, el nombre del archivo de salida. */
            nombre_archivo_salida = optarg;
            rotura;

        caso 'v': /* -v o --verbose */
            detallado = 1;
            rotura;

        case '?': /* El usuario especificó una opción no válida. */
            /* Imprime la información de uso en el error estándar y sale con exit
            código uno (que indica una terminación anormal). */
            print_usage (stderr, 1);

        case -1: /* Hecho con opciones. */
            rotura;

        predeterminado: /* Algo más: inesperado. */
            abortar ();
    }
}
while (next_option != -1);

/* Hecho con opciones. OPTIND apunta al primer argumento de no opción.
Para fines de demostración, imprímalos si la opción detallada era int i;
especificado. */
if (detallado) {
    para (i = optind; i < argc; ++ i)
        printf ("Argumento:%s\n", argv [i]);
}

/* El programa principal va aquí. */ return 0;

}

```

Utilizando `getopt_long` puede parecer mucho trabajo, pero escribir código para analizar las opciones de la línea de comandos por sí mismo tomaría aún más tiempo. `getopt_long` La función es muy sofisticada y permite una gran flexibilidad para especificar qué tipo de opciones aceptar. Sin embargo, es una buena idea mantenerse alejado de las funciones más avanzadas y ceñirse a la estructura básica de opciones descrita.

2.1.4 E / S estándar

La biblioteca C estándar proporciona flujos de entrada y salida estándar (`stdin` y `stdout`, respectivamente). Estos son utilizados por `scanf`, `printf`, y otras funciones de la biblioteca. En la tradición de UNIX, el uso de entrada y salida estándar es habitual para los programas GNU / Linux. Esto permite el encadenamiento de múltiples programas usando tuberías de shell y redireccionamiento de entrada y salida. (Consulte la página del manual de su shell para conocer su sintaxis).

La biblioteca C también proporciona `stderr`, el flujo de error estándar. Los programas deben imprimir mensajes de advertencia y error como error estándar en lugar de la salida estándar. Esto permite a los usuarios separar la salida normal y los mensajes de error, por ejemplo, redirigiendo la salida estándar a un archivo mientras permite que el error estándar se imprima en la consola. `fprintf` La función se puede utilizar para imprimir en `stderr`, por ejemplo:

```
fprintf(stderr, ("Error: ..."));
```

Estos tres flujos también son accesibles con los comandos de E / S de UNIX subyacentes (`leer`, `escribir`, y así sucesivamente) a través de descriptores de archivo. Estos son descriptores de archivo 0 para `stdin`, 1 para `stdout`, y 2 para `stderr`.


Al invocar un programa, a veces es útil redirigir tanto la salida estándar como el error estándar a un archivo o canalización. La sintaxis para hacer esto varía entre shells; para carcasas estilo Bourne (incluidas `intento`, el shell predeterminado en la mayoría de las distribuciones GNU / Linux), la sintaxis es la siguiente:

```
% programa> archivo_salida.txt 2> & 1%
programa 2> & 1 | filtrar
```

los `2> y 1` la sintaxis indica que el descriptor de archivo 2 (`stderr`) debe fusionarse en el descriptor de archivo 1 (`stdout`). Tenga en cuenta que `2> y 1` debe seguir una redirección de archivos (el primer ejemplo) pero debe preceder a una redirección de tubería (el segundo ejemplo).

Tenga en cuenta que `stdout` está almacenado en búfer. Datos escritos en `stdout` no se envía a la consola (u otro dispositivo, si es redirigido) hasta que el búfer se llena, el programa sale normalmente, o `stdout` está cerrado. Puede vaciar explícitamente el búfer llamando a lo siguiente:

```
fflush (salida estándar);
```

A diferencia de, `stderr` no está almacenado en búfer; datos escritos en `stderr` va directamente a la consola. 

^[1]En C ++, la misma distinción se aplica a `cout` y `cerr`, respectivamente. Tenga en cuenta que el `endl` el token vacía una secuencia además de imprimir un carácter de nueva línea; si no desea vaciar la secuencia (por razones de rendimiento, por ejemplo), use una constante de nueva línea, `'\n'`, en lugar de.

Esto puede producir algunos resultados sorprendentes. Por ejemplo, este ciclo no imprime un punto por segundo; en su lugar, los puntos se almacenan en búfer y un montón de ellos se imprimen juntos cuando el búfer se llena.

```
while (1) {
    printf (".");
    dormir (1);
}
```

```
}
```

En este ciclo, sin embargo, los puntos aparecen una vez por segundo:

```
while (1) {  
    fprintf(stderr, ".");  
    dormir (1);  
}
```

2.1.5 Códigos de salida del programa

Cuando un programa finaliza, indica su estado con un código de salida. El código de salida es un pequeño entero; por convención, un código de salida de cero denota una ejecución exitosa, mientras que los códigos de salida distintos de cero indican que ocurrió un error. Algunos programas utilizan diferentes valores de código de salida distintos de cero para distinguir errores específicos.

Con la mayoría de los shells, es posible obtener el código de salida del programa ejecutado más recientemente usando la función especial `PS` variable. He aquí un ejemplo en el que `ls` El comando se invoca dos veces y su código de salida se imprime después de cada invocación. En el primer caso, `ls` se ejecuta correctamente y devuelve el código de salida cero. En el segundo caso, `ls` encuentra un error (porque el nombre de archivo especificado en la línea de comando no existe) y, por lo tanto, devuelve un código de salida distinto de cero.

```
% ls /  
compartimiento coda etc lib          misc nfs proc sbin usr  
boot dev home perdido + encontrado mnt opt  root tmp var%  
echo $?  
0  
% ls archivo falso  
ls: bogusfile: ¿No existe ese archivo o directorio%  
echo $?  
1
```

El programa AC o C ++ especifica su código de salida devolviendo ese valor del `principal` función. Existen otros métodos para proporcionar códigos de salida y se asignan códigos de salida especiales a los programas que terminan de forma anormal (mediante una señal). Estos se analizan con más detalle en [Capítulo 3](#).

2.1.6 El medio ambiente

GNU / Linux proporciona a cada programa en ejecución una *medio ambiente*. El entorno es una colección de pares de variable / valor. Tanto los nombres de las variables de entorno como sus valores son cadenas de caracteres. Por convención, los nombres de las variables de entorno se escriben en mayúsculas.

Probablemente ya esté familiarizado con varias variables de entorno comunes. Por ejemplo:

- `USUARIO` contiene su nombre de usuario.
- `HOGAR` contiene la ruta a su directorio personal.
- `SENDERO` contiene una lista de directorios separados por dos puntos a través de los cuales Linux busca los comandos que usted invoca.
- `MONITOR` contiene el nombre y el número de visualización del servidor X Window en el que aparecerán las ventanas de los programas gráficos X Window.

Su shell, como cualquier otro programa, tiene un entorno. Las conchas proporcionan métodos para examinar y modificar el medio ambiente directamente. Para imprimir el entorno actual en su shell, invoque el `printenv` programa. Varios shells tienen una sintaxis incorporada diferente para usar variables de entorno; la siguiente es la sintaxis de los shells de estilo Bourne.

- El shell crea automáticamente una variable de shell para cada variable de entorno que encuentra, por lo que puede acceder a los valores de las variables de entorno utilizando el `PSvarname` sintaxis. Por ejemplo:
 - `% echo $ USER`
 - `Samuel`
 - `% echo $ HOME`
`/ inicio / samuel`
- Puedes usar el `exportar` comando para exportar una variable de shell al entorno. Por ejemplo, para configurar el `EDITOR` variable de entorno, usaría esto:
 - `% EDITOR = emacs`
`% de exportación EDITOR`

O, para abreviar:

```
% export EDITOR = emacs
```

En un programa, accede a una variable de entorno con la `getenv` funcionar en `<stdlib.h>`. Esa función toma un nombre de variable y devuelve el valor correspondiente como una cadena de caracteres, o `NULO` si esa variable no está definida en el entorno. Para establecer o borrar las variables de entorno, utilice el `setenv` y `unsetenv` funciones, respectivamente.

Enumerar todas las variables del entorno es un poco más complicado. Para hacer esto, debe acceder a una variable global especial llamada `reinar`, que se define en la biblioteca GNU C. Esta variable, de tipo `carbonizarse**`, es un `NULO` -matriz terminada de punteros a cadenas de caracteres. Cada cadena contiene una variable de entorno, en la forma `VARIABLE = valor`.

El programa en [Listado 2.3](#), por ejemplo, simplemente imprime todo el entorno haciendo un bucle a través del `reinar` formación.

Listado 2.3 (print-env.c) Impresión del entorno de ejecución

```
# incluye <stdio.h>
```

```
/ * La variable ENVIRON contiene el entorno. * / extern char ** entorno;
```

```
int main ()
{
    char ** var;
    para (var = ambiente; * var! = NULL; ++ var)
        printf ("%s \n", * var);
    return 0;
}
```

No modifiques `reinar` tú mismo; utilizar el `setenv` y `unsetenv` funciones en su lugar.

Normalmente, cuando se inicia un programa nuevo, hereda una copia del entorno del programa que lo invocó (el programa shell, si se invocó de forma interactiva). Entonces, por ejemplo, los programas que ejecuta desde el shell pueden examinar los valores de las variables de entorno que establece en el shell.

Las variables de entorno se utilizan comúnmente para comunicar información de configuración a los programas. Supongamos, por ejemplo, que está escribiendo un programa que se conecta a un servidor de Internet para obtener alguna información. Puede escribir el programa de modo que el nombre del servidor se especifique en la línea de comandos. Sin embargo, suponga que el nombre del servidor no es algo que los usuarios cambien con mucha frecuencia. Puede utilizar una variable de entorno especial, digamos `NOMBRE DEL SERVIDOR`—Para especificar el nombre del servidor; si esa variable no existe, se utiliza un valor predeterminado. Parte de su programa puede verse como se muestra en [Listado 2.4](#).

Listado 2.4 (*client.c*) Parte de un programa de cliente de red

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

int main ()
{
    char * nombre_servidor = getenv ("NOMBRE_Servidor"); si
    (nombre_servidor == NULL)
        /* No se estableció la variable de entorno SERVER_NAME. Utilizar el
           defecto. */
        nombre_servidor = "servidor.miempresa.com";

    printf ("accediendo al servidor% s \n", nombre_servidor); /*
    Accede al servidor aquí ... */

    return 0;
}
```

Supongamos que este programa se llama `cliente`. Suponiendo que no ha configurado el `NOMBRE DEL SERVIDOR` variable, se utiliza el valor predeterminado para el nombre del servidor:

```
% cliente
accediendo al servidor server.my-company.com
```

Pero es fácil especificar un servidor diferente:

```
% export SERVER_NAME = backup-server.elsewhere.net% client
accediendo al servidor backup-server.elsewhere.net
```

2.1.7 Uso de archivos temporales

A veces, un programa necesita crear un archivo temporal, almacenar datos grandes durante un tiempo o pasar datos a otro programa. En los sistemas GNU / Linux, los archivos temporales se almacenan en el `/tmp` directorio. Al utilizar archivos temporales, debe tener en cuenta los siguientes errores:

- Se puede ejecutar más de una instancia de su programa simultáneamente (por el mismo usuario o por diferentes usuarios). Las instancias deben usar diferentes nombres de archivos temporales para que no colisionen.
- Los permisos de archivo del archivo temporal deben establecerse de tal manera que los usuarios no autorizados no puedan alterar la ejecución del programa modificando o reemplazando el archivo temporal.
- Los nombres de archivos temporales deben generarse de manera que no se puedan predecir externamente; de lo contrario, un atacante puede aprovechar la demora entre probar si un nombre ya está en uso y abrir un nuevo archivo temporal.

GNU / Linux proporciona funciones, [mkstemp](#) y [tmpfile](#), que se encargan de estos problemas por usted (además de varias funciones que no lo hacen). El que use depende de si planea entregar el archivo temporal a otro programa y si desea usar UNIX I / O ([abierto](#), [escribir](#), y así sucesivamente) o las funciones de E / S de flujo de la biblioteca C ([fopen](#), [fprintf](#), etcétera).

Usando mkstemp

los [mkstemp](#) La función crea un nombre de archivo temporal único a partir de una plantilla de nombre de archivo, crea el archivo con permisos para que solo el usuario actual pueda acceder a él y abre el archivo para lectura / escritura. La plantilla de nombre de archivo es una cadena de caracteres que termina en "XXXXXX" (seis X mayúsculas); [mkstemp](#) reemplaza las X con caracteres para que el nombre del archivo sea único. El valor de retorno es un descriptor de archivo; utilizar [escribir](#) familia de funciones para escribir en el archivo temporal.

Archivos temporales creados con [mkstemp](#) no se eliminan automáticamente. Depende de usted eliminar el archivo temporal cuando ya no sea necesario. (Los programadores deben tener mucho cuidado al limpiar los archivos temporales; de lo contrario, el [tmp](#) el sistema de archivos se llenará eventualmente, dejando el sistema inoperable.) Si el archivo temporal es solo para uso interno y no se entregará a otro programa, es una buena idea llamar [desconectar](#) en el archivo temporal inmediatamente. los [desconectar](#) La función elimina la entrada de directorio correspondiente a un archivo, pero debido a que los archivos en un sistema de archivos se cuentan por referencias, el archivo en sí no se elimina hasta que no haya descriptores de archivo abiertos para ese archivo. De esta manera, su programa puede continuar usando el archivo temporal y el archivo desaparece automáticamente tan pronto como cierra el descriptor de archivo. Debido a que Linux cierra los descriptores de archivos cuando finaliza un programa, el archivo temporal se eliminará incluso si su programa termina de forma anormal.

El par de funciones en [Listado 2.5](#) demuestra [mkstemp](#). Utilizadas juntas, estas funciones facilitan la escritura de un búfer de memoria en un archivo temporal (para que la memoria se pueda liberar o reutilizar) y luego volver a leerla más tarde.

Listado 2.5 (temp_file.c) Usando mkstemp

```
# incluye <stdlib.h>
# incluye <unistd.h>

/* Un identificador para un archivo temporal creado con write_temp_file. En
esta implementación, es solo un descriptor de archivo. */ typedef int
temp_file_handle;

/* Escribe LENGTH bytes desde BUFFER en un archivo temporal. los
El archivo temporal se desvincula inmediatamente. Devuelve un identificador al
archivo temporal. */

temp_file_handle write_temp_file (char * buffer, size_t length) {

    /* Crea el nombre del archivo y el archivo. El XXXXXX será reemplazado por
caracteres que hacen que el nombre de archivo sea único.
*/ char temp_filename [] = "/tmp/temp_file.XXXXXX"; int fd =
mkstemp (temp_filename);
/* Desvincular el archivo inmediatamente, para que se elimine cuando el
el descriptor de archivo está cerrado. */
unlink (temp_filename);
/* Escriba primero el número de bytes en el archivo. */ write (fd, &
length, sizeof (length));
/* Ahora escribe los datos en sí. */ write
(fd, búfer, longitud);
/* Utilice el descriptor de archivo como identificador del archivo temporal. */ return fd;
}

/* Lee el contenido de un archivo temporal TEMP_FILE creado con
```

`write_temp_file`. El valor de retorno es un búfer recién asignado de esos contenidos, que la persona que llama debe desasignar de forma gratuita.
* `LENGTH` se establece en el tamaño del contenido, en bytes. Se elimina el archivo temporal. */

```
char * read_temp_file (temp_file_handle temp_file, size_t * length) {  
  
    char * buffer;  
    /* El identificador TEMP_FILE es un descriptor de archivo para el archivo temporal. */ int fd =  
    archivo_temp;  
    /* Rebobinar hasta el principio del archivo. */ lseek (fd,  
    0, SEEK_SET);  
    /* Leer el tamaño de los datos en el archivo temporal. */ leer (fd,  
    longitud, tamaño de (* longitud));  
    /* Asignar un búfer y leer los datos. */ buffer = (char *)  
    malloc (* longitud);  
    leer (fd, búfer, * longitud);  
    /* Cierre el descriptor de archivo, lo que hará que el archivo temporal se  
        irse. */  
    cerrar (fd);  
    búfer de retorno;  
}
```

Usando tmpfile

Si está utilizando las funciones de E / S de la biblioteca C y no necesita pasar el archivo temporal a otro programa, puede utilizar el `tmpfile` función. Esto crea y abre un archivo temporal y le devuelve un puntero de archivo. El archivo temporal ya está desvinculado, como en el ejemplo anterior, por lo que se elimina automáticamente cuando se cierra el puntero del archivo (con `fclose`) o cuando el programa termina.

GNU / Linux proporciona varias otras funciones para generar archivos temporales y nombres de archivos temporales, que incluyen `mktemp`, `tmpnam`, y `tempnam`. Sin embargo, no use estas funciones porque sufren los problemas de confiabilidad y seguridad ya mencionados.

2.2 Codificación defensiva

Escribir programas que se ejecuten correctamente con un uso "normal" es difícil; escribir programas que se comporten con elegancia en situaciones de falla es más difícil. Esta sección muestra algunas técnicas de codificación para encontrar errores en forma temprana y para detectar y recuperarse de problemas en un programa en ejecución.

Los ejemplos de código que se presentan más adelante en este libro omiten deliberadamente un extenso código de verificación y recuperación de errores porque esto oscurecería la funcionalidad básica que se presenta. Sin embargo, el ejemplo final en [Capítulo 11](#), "Una aplicación GNU / Linux de muestra", vuelve a demostrar cómo usar estas técnicas para escribir programas robustos.

2.2.1 Uso *afirmar*

Un buen objetivo a tener en cuenta al codificar programas de aplicación es que los errores o errores inesperados deberían hacer que el programa falle de forma espectacular, lo antes posible. Esto le ayudará a encontrar errores al principio de los ciclos de desarrollo y prueba. Las fallas que no se manifiestan dramáticamente a menudo se pasan por alto y no aparecen hasta que la aplicación está en manos de los usuarios.

Uno de los métodos más simples para verificar condiciones inesperadas es el estándar C `assert` macro. El argumento de esta macro es una expresión booleana. El programa finaliza si la expresión se evalúa como falsa, después de imprimir un mensaje de error que contiene el archivo de origen, el número de línea y el texto de la expresión. La macro de aserción es muy útil para una amplia variedad de comprobaciones de coherencia

interno a un programa. Por ejemplo, use `afirmar` para probar la validez de los argumentos de la función, para probar las condiciones previas y posteriores de las llamadas a funciones (y llamadas a métodos, en C++) y para probar los valores de retorno inesperados.

Cada uso de `afirmar` sirve no solo como una verificación en tiempo de ejecución de una condición, sino también como documentación sobre el funcionamiento del programa dentro del código fuente. Si su programa contiene un `afirmar` (*condición*) que le dice a alguien que lee tu código fuente que *condición* siempre debe ser cierto en ese punto del programa, y si *condición* no es cierto, probablemente sea un error en el programa.

Para el código de rendimiento crítico, las comprobaciones en tiempo de ejecución, como los usos de `afirmar` puede imponer una penalización de rendimiento significativa. En estos casos, puede compilar su código con el `NDEBUG` macro definida, utilizando el `-DNDEBUG` bandera en la línea de comando del compilador. Con `NDEBUG` conjunto, apariciones del `afirmar` La macro se preprocesará. Sin embargo, es una buena idea hacer esto solo cuando sea necesario por razones de rendimiento, y solo con archivos de origen críticos para el rendimiento.

Porque es posible preprocesar `afirmar` macros, tenga cuidado de que cualquier expresión que utilice con `afirmar` no tiene efectos secundarios. Específicamente, no debes llamar a funciones dentro `afirmar` expresiones, asignar variables o utilizar operadores de modificación como `++`. Suponga, por ejemplo, que llama a una función, `hacer algo`, repetidamente en un bucle. `hacer algo` La función devuelve cero en caso de éxito y distinto de cero en caso de error, pero no espera que nunca falle en su programa. Podría tener la tentación de escribir:

```
para (i = 0; i < 100; ++ i)
    afirmar (hacer_algo () == 0);
```

Sin embargo, es posible que descubra que esta verificación en tiempo de ejecución impone una penalización de rendimiento demasiado grande y luego decida volver a compilar con `NDEBUG` definido. Esto eliminará el `afirmar` llamar por completo, por lo que la expresión nunca se evaluará y `hacer algo` nunca será llamado. Deberías escribir esto en su lugar:

```
para (i = 0; i < 100; ++ i) {
    int status = hacer_algo (); afirmar
    (estado == 0);
}
```

Otra cosa a tener en cuenta es que no debes usar `afirmar` para probar la entrada de usuario no válida. A los usuarios no les gusta cuando las aplicaciones simplemente se bloquean con un mensaje de error críptico, incluso en respuesta a una entrada no válida. Aún así, siempre debe verificar si hay entradas no válidas y producir mensajes de error sensibles en la entrada de respuesta. Usar `afirmar` solo para comprobaciones internas en tiempo de ejecución.

Algunos buenos lugares para usar `afirmar` son estos:

- Verifique con punteros nulos, por ejemplo, como argumentos de función no válidos. El mensaje de error generado por `{afirmar (puntero! = NULO)}`,

La afirmación '`puntero! = ((Void *) 0)`' falló.

es más informativo que el mensaje de error que resultaría si su programa desreferenciara un puntero nulo:

Fallo de segmentación (núcleo volcado)

- Compruebe las condiciones de los valores de los parámetros de función. Por ejemplo, si una función debe llamarse solo con un valor positivo para el parámetro `foo`, use esto al comienzo del cuerpo de la función:

afirmar (foo > 0);

Esto le ayudará a detectar usos incorrectos de la función, y también le dejará muy claro a alguien que lea el código fuente de la función que existe una restricción en el valor del parámetro.

No te reprimas; usar `afirmar` generosamente a lo largo de sus programas.

2.2.2 Fallos de llamadas al sistema

A la mayoría de nosotros se nos enseñó originalmente cómo escribir programas que se ejecutan hasta su finalización a lo largo de una ruta bien definida. Dividimos el programa en tareas y subtareas, y cada función completa una tarea invocando otras funciones para realizar las subtareas correspondientes. Dadas las entradas apropiadas, esperamos que una función produzca la salida y los efectos secundarios correctos.

Las realidades del hardware y software de las computadoras se entrometen en este sueño idealizado. Las computadoras tienen recursos limitados; el hardware falla; muchos programas se ejecutan al mismo tiempo; los usuarios y programadores cometen errores. A menudo, es en el límite entre la aplicación y el sistema operativo donde se muestran estas realidades. Por lo tanto, cuando se utilizan llamadas al sistema para acceder a los recursos del sistema, para realizar E / S o para otros fines, es importante comprender no solo qué sucede cuando la llamada se realiza correctamente, sino también cómo y cuándo puede fallar la llamada.

Las llamadas al sistema pueden fallar de muchas formas. Por ejemplo:

- El sistema puede quedarse sin recursos (o el programa puede exceder los límites de recursos impuestos por el sistema de un solo programa). Por ejemplo, el programa puede intentar asignar demasiada memoria, escribir demasiada en un disco o abrir demasiados archivos al mismo tiempo.
- Linux puede bloquear una determinada llamada al sistema cuando un programa intenta realizar una operación para la que no tiene permiso. Por ejemplo, un programa puede intentar escribir en un archivo marcado como de solo lectura, acceder a la memoria de otro proceso o eliminar el programa de otro usuario.
- Los argumentos de una llamada al sistema pueden no ser válidos, ya sea porque el usuario proporcionó una entrada no válida o debido a un error del programa. Por ejemplo, el programa puede pasar una dirección de memoria no válida o un descriptor de archivo no válido a una llamada al sistema. O un programa puede intentar abrir un directorio como un archivo ordinario, o puede pasar el nombre de un archivo normal a una llamada al sistema que espera un directorio.
- Una llamada al sistema puede fallar por motivos externos a un programa. Esto sucede con mayor frecuencia cuando una llamada al sistema accede a un dispositivo de hardware. Es posible que el dispositivo esté defectuoso o que no admita una operación en particular, o quizás no haya un disco insertado en la unidad.
- En ocasiones, una llamada al sistema puede ser interrumpida por un evento externo, como la entrega de una señal. Es posible que esto no indique una falla absoluta, pero es responsabilidad del programa que realiza la llamada reiniciar la llamada al sistema, si lo desea.

En un programa bien escrito que hace un uso extensivo de las llamadas al sistema, a menudo ocurre que se dedica más código a detectar y manejar errores y otras circunstancias excepcionales que al trabajo principal del programa.

2.2.3 Códigos de error de llamadas al sistema

La mayoría de las llamadas al sistema devuelven cero si la operación tiene éxito, o un valor distinto de cero si la operación falla. (Muchos, sin embargo, tienen diferentes convenciones de valor de retorno; por ejemplo, `malloc` devuelve un puntero nulo para indicar falla. Siempre lea atentamente la página de manual cuando utilice una llamada al sistema.) Aunque esta información puede ser suficiente para determinar si el programa debe continuar ejecutándose como de costumbre, probablemente no proporcione suficiente información para una recuperación sensata de errores.

La mayoría de las llamadas al sistema usan una variable especial llamada `errno` para almacenar información adicional en caso de falla. Cuando falla una llamada, el sistema configura `errno` a un valor que indique qué salió mal. Porque todas las llamadas al sistema usan el mismo `errno` variable para almacenar información de error, debe copiar el valor en otra variable inmediatamente después de la llamada fallida. El valor de `errno` se sobrescribirá la próxima vez que realice una llamada al sistema.

[2] En realidad, por razones de seguridad de subprocessos, `errno` se implementa como una macro, pero se usa como una variable global.

Los valores de error son números enteros; los valores posibles están dados por macros de preprocesador, por convención nombrados en mayúsculas y comenzando con "E", por ejemplo, `EACCES` y `EINVAL`. Utilice siempre estas macros para hacer referencia a `errno` valores en lugar de valores enteros. Incluir la `<errno.h>` encabezado si usa `errno` valores.

GNU / Linux proporciona una función conveniente, `strerror`, que devuelve una descripción de cadena de caracteres de un `errno` código de error, adecuado para su uso en mensajes de error. Incluir `<string.h>` si utiliza `strerror`

GNU / Linux también proporciona `perror`, que imprime la descripción del error directamente en el `stderr` Arroyo. Pasar `aperror` un prefijo de cadena de caracteres para imprimir antes de la descripción del error, que normalmente debe incluir el nombre de la función que falló. Incluir `<stdio.h>` si utiliza `perror`.

Este fragmento de código intenta abrir un archivo; si la apertura falla, imprime un mensaje de error y sale del programa. Tenga en cuenta que `elabierto` call devuelve un descriptor de archivo abierto si la operación de apertura tiene éxito, o -1 si la operación falla.

```
fd = open ("archivo de entrada.txt", O_RDONLY); si
(fd == -1) {
    /* La apertura falló. Imprima un mensaje de error y salga. */ fprintf (stderr, "error
    al abrir el archivo:% s \ n", strerror (errno)); salida (1);
}
```

Dependiendo de su programa y la naturaleza de la llamada al sistema, la acción apropiada en caso de falla podría ser imprimir un mensaje de error, cancelar una operación, abortar el programa, volver a intentarlo o incluso ignorar el error. Sin embargo, es importante incluir lógica que maneje todos los posibles modos de falla de una forma u otra.

Un posible código de error por el que debería estar atento, especialmente con las funciones de E / S, es `EINTR`. Algunas funciones, como `leer`, `Seleccione`, y `dormir`, puede llevar un tiempo considerable para ejecutarse. Estos son considerados *bloqueo* funciona porque la ejecución del programa está bloqueada hasta que se completa la llamada. Sin embargo, si el programa recibe una señal mientras está bloqueado en una de estas llamadas, la llamada regresará sin completar la operación. En este caso, `errno` se establece en `EINTR`. Por lo general, querrá volver a intentar la llamada al sistema en este caso.

Aquí hay un fragmento de código que usa el `chown` llamar para cambiar el propietario de un archivo dado por `sendero` al usuario por `user_id`. Si la llamada falla, el programa toma acción dependiendo del valor de `errno`. Observe que cuando detectamos lo que probablemente sea un error en el programa, salimos usando `abortar` o `afirmar`, cuales

hacer que se genere un archivo central. Esto puede resultar útil para la depuración post mortem. Para otros errores irreversibles, como condiciones de falta de memoria, salimos usando `Salida` y un valor de salida distinto de cero en su lugar porque un archivo central no sería muy útil.

```
rval = chown (ruta, id_usuario, -1); if (rval !=
0) {
    /* Guarde errno porque la próxima llamada al sistema lo aplastará. */ int error_code =
    errno;
    /* La operación no tuvo éxito; chown debería devolver -1 en caso de error. */ aaseverar (rval
    == -1);
    /* Verifique el valor de errno y tome la acción apropiada. */ switch (código_error)
    {
        case EPERM: /* Permiso denegado. */
        case EROFS: /* PATH está en un sistema de archivos de solo lectura. */
        case ENAMETOOLONG: /* PATH es demasiado largo. */ case
        ENOENT: /* PATH no sale. */
        case ENOTDIR: /* Un componente de PATH no es un directorio. */
        case EACCES: /* No se puede acceder a un componente de PATH. */
            /* Algo anda mal con el archivo. Imprime un mensaje de error. */ fprintf (stderr,
            "error al cambiar la propiedad de %s: %s\n",
                ruta, strerror (código_error));
            /* No finalice el programa; tal vez darle al usuario la oportunidad de
            elegir otro archivo ... */
            rotura;

        caso EFAULT:
            /* PATH contiene una dirección de memoria no válida. Probablemente sea un error. */ abortar
            ();

        caso ENOMEM:
            /* Se quedó sin memoria del kernel. */
            fprintf (stderr, "%s\n", strerror (código_error)); salida (1);

        defecto:
            /* Algún otro código de error inesperado. Hemos tratado de manejar todo
            posibles códigos de error; si nos hemos perdido uno, ¡es un error! */ abortar ();

    };
}
```

Simplemente podría haber usado este código, que se comporta de la misma manera si la llamada tiene éxito:

```
rval = chown (ruta, id_usuario, -1); afirmar
(rval == 0);
```

Pero si la llamada falla, esta alternativa no hace ningún esfuerzo por informar, manejar o recuperarse de los errores.

Si utiliza el primer formulario, el segundo formulario o algo intermedio, depende de los requisitos de detección y recuperación de errores de su programa.

2.2.4 Errores y asignación de recursos

A menudo, cuando falla una llamada al sistema, es apropiado cancelar la operación actual pero no terminar el programa porque es posible recuperarse del error. Una forma de hacer esto es regresar de la función actual, pasando un código de retorno a la persona que llama indicando el error.

Si decide regresar desde la mitad de una función, es importante asegurarse de que todos los recursos asignados con éxito previamente en la función se desasignen primero. Estos recursos pueden incluir

memoria, descriptores de archivo, punteros de archivo, archivos temporales, objetos de sincronización, etc. De lo contrario, si su programa continúa ejecutándose, se filtrarán los recursos asignados antes de que ocurriera la falla.

Considere, por ejemplo, una función que lee de un archivo a un búfer. La función puede seguir estos pasos:

1. Asigne el búfer.
2. Abra el archivo.
3. Leer del archivo en el búfer.
4. Cierre el archivo.
5. Devuelva el búfer.

Si el archivo no existe, el paso 2 fallará. Un curso de acción apropiado podría ser regresar **NULL** de la función. Sin embargo, si el búfer ya se ha asignado en el paso 1, existe el riesgo de que se pierda esa memoria. Debe recordar desasignar el búfer en algún lugar a lo largo de cualquier flujo de control del que no regrese. Si el paso 3 falla, no solo debe desasignar el búfer antes de regresar, sino que también debe cerrar el archivo.

[Listado 2.6](#) muestra un ejemplo de cómo podría escribir esta función.

Listado 2.6 (readfile.c) Liberar recursos durante condiciones anormales

```
# incluye <fcntl.h>
# incluye <stdlib.h>
# incluir <sys / stat.h>
# incluye <sys / types.h>
# incluye <unistd.h>
char * read_from_file (const char * nombre de archivo, tamaño_t longitud) {

    char * buffer;
    int fd;
    ssize_t bytes_read;

    /* Asignar el búfer. */ buffer = (char *)
    malloc (longitud); si (buffer == NULL)

        return NULL;
    /* Abre el archivo. */
    fd = open (nombre de archivo,
    O_RDONLY); si (fd == -1) {
        /* apertura fallida. Desasigne el búfer antes de regresar. */ free (búfer);

        return NULL;
    }
    /* Leer los datos. */
    bytes_read = leer (fd, búfer, longitud);
    if (bytes_read != longitud) {
        /* lectura fallida. Desasigne el búfer y cierre fd antes de regresar. */ free (búfer);

        cerrar (fd);
        return NULL;
    }
    /* Todo está bien. Cierre el archivo y devuelva el búfer. */
    cerrar (fd);
```



```
    búfer de retorno;  
}
```

Linux limpia la memoria asignada, los archivos abiertos y la mayoría de los demás recursos cuando un programa termina, por lo que no es necesario desasignar búferes y cerrar archivos antes de llamar [Salida](#). Sin embargo, es posible que deba liberar manualmente otros recursos compartidos, como archivos temporales y memoria compartida, que potencialmente pueden sobrevivir a un programa.

2.3 Escritura y uso de bibliotecas

Prácticamente todos los programas están vinculados a una o más bibliotecas. Cualquier programa que use una función C (como [printf](#) o [malloc](#)) se vinculará con la biblioteca de tiempo de ejecución de C. Si su programa tiene una interfaz gráfica de usuario (GUI), se vinculará con las bibliotecas de ventanas. Si su programa utiliza una base de datos, el proveedor de la base de datos le proporcionará bibliotecas que puede utilizar para acceder a la base de datos cómodamente.

En cada uno de estos casos, debe decidir si vincular la biblioteca *inactivamente* o *dinamicamente*. Si elige vincular estáticamente, sus programas serán más grandes y más difíciles de actualizar, pero probablemente más fáciles de implementar. Si se vincula dinámicamente, sus programas serán más pequeños, más fáciles de actualizar, pero más difíciles de implementar. Esta sección explica cómo vincular tanto estática como dinámicamente, examina las compensaciones con más detalle y brinda algunas "reglas generales" para decidir qué tipo de vinculación es mejor para usted.

2.3.1 Archivos

Un *archivo* (biblioteca estática) es simplemente una colección de archivos de objetos almacenados como un solo archivo. (Un archivo es aproximadamente el equivalente a un Windows [.LIB](#) file.) Cuando proporciona un archivo al vinculador, el vinculador busca en el archivo los archivos de objeto que necesita, los extrae y los vincula a su programa como si hubiera proporcionado esos archivos de objeto directamente.

Puede crear un archivo utilizando el [Arkansas](#) mando. Los archivos de archivo tradicionalmente usan un [.a](#) extensión en lugar de la [.o](#) extensión utilizada por archivos de objeto ordinarios. Así es como combinarías [test1.o](#) y [test2.o](#) en un solo [libtest.a](#) archivo:

```
% ar cr libtest.a test1.o test2.o
```

los [cr](#) banderas dicen [Arkansas](#) para crear el archivo. Ahora puede vincular con este archivo utilizando el [-ltest](#) opción con [gcc](#) o [g++](#), como se describe en [Sección 1.2.5](#), "Vinculación de archivos de objeto", en [Capítulo 1](#), "Empezando."

^[1] Puede utilizar otros indicadores para eliminar un archivo de un archivo o para realizar otras operaciones en el archivo. Estas operaciones rara vez se utilizan pero están documentadas en el

[Arkansas](#) página de manual.

Cuando el vinculador encuentra un archivo en la línea de comando, busca en el archivo todas las definiciones de símbolos (funciones o variables) a las que se hace referencia en los archivos de objeto que ya ha procesado pero que aún no ha definido. Los archivos de objeto que definen esos símbolos se extraen del archivo y se incluyen en el ejecutable final. Debido a que el enlazador busca en el archivo cuando se encuentra en la línea de comando, generalmente tiene sentido colocar archivos al final de la línea de comando. Por ejemplo, suponga que [prueba.c](#) contiene el código en [Listado 2.7](#) y [app.c](#) contiene el código en [Listado 2.8](#).

Listado 2.7 (test.c) Contenido de la biblioteca

```
int f ()  
{  
    volver 3;  
}
```

```
}
```

Listado 2.8 (app.c) Un programa que usa funciones de biblioteca

```
int main ()  
{  
    return f ();  
}
```

Ahora suponga que `test.o` se combina con algunos otros archivos de objeto para producir el `libtest.a` archivo. La siguiente línea de comando no funcionará:

```
% gcc -o aplicación -L. -ltest app.o app.o: En la función  
'main': app.o (.text + 0x4): referencia indefinida a 'f'  
collect2: ld devolvió 1 estado de salida
```

El mensaje de error indica que aunque `libtest.a` contiene una definición de `F`, el enlazador no lo encontró. Eso es porque `libtest.a` se buscó cuando se encontró por primera vez, y en ese momento el enlazador no había visto ninguna referencia a `F`.

Por otro lado, si usamos esta línea, no se emiten mensajes de error:

```
% gcc -o aplicación aplicación.o -L. -Ltest
```

La razón es que la referencia a `F` en `app.o` hace que el enlazador incluya el `test.o` archivo de objeto del `libtest.a` archivo.

2.3.2 Bibliotecas compartidas

A *biblioteca compartida* también conocido como un objeto compartido, o como una biblioteca enlazada dinámicamente) es similar a un archivo en que es una agrupación de archivos de objeto. Sin embargo, existen muchas diferencias importantes. La diferencia más fundamental es que cuando una biblioteca compartida está vinculada a un programa, el ejecutable final en realidad no contiene el código que está presente en la biblioteca compartida. En cambio, el ejecutable simplemente contiene una referencia a la biblioteca compartida. Si varios programas del sistema están vinculados a la misma biblioteca compartida, todos harán referencia a la biblioteca, pero en realidad no se incluirá ninguno. De esta forma, la biblioteca se "comparte" entre todos los programas que la enlazan.

Una segunda diferencia importante es que una biblioteca compartida no es simplemente una colección de archivos de objeto, de los cuales el enlazador elige aquellos que son necesarios para satisfacer las referencias no definidas. En cambio, los archivos de objeto que componen la biblioteca compartida se combinan en un solo archivo de objeto de modo que un programa que se vincula con una biblioteca compartida siempre incluye todo el código de la biblioteca, en lugar de solo las partes que se necesitan.

Para crear una biblioteca compartida, debe compilar los objetos que compondrán la biblioteca utilizando el `-fPIC` opción al compilador, así:

```
% gcc -c -fPIC test1.c
```

los `-fPIC` La opción le dice al compilador que vas a usar `test.o` como parte de un objeto compartido.

Código independiente de la posición (PIC)

PIC significa código independiente de la posición. Las funciones de una biblioteca compartida pueden cargarse en diferentes direcciones en diferentes programas, por lo que el código del objeto compartido no debe depender de la dirección (o posición) en la que se carga. Esta consideración no tiene ningún impacto en usted, como programador, excepto que debe recordar utilizar el `-fPIC` marca al compilar código que se utilizará en una biblioteca compartida.

Luego, combina los archivos de objeto en una biblioteca compartida, como esta:

```
% gcc -shared -fPIC -o libtest.so test1.o test2.o
```

los `-compartido` La opción le dice al enlazador que produzca una biblioteca compartida en lugar de un ejecutable ordinario. Las bibliotecas compartidas usan la extensión `.so`, que significa objeto compartido. Al igual que los archivos estáticos, el nombre siempre comienza con `lib` para indicar que el archivo es una biblioteca.

Vincular con una biblioteca compartida es como vincular con un archivo estático. Por ejemplo, la siguiente línea se vinculará con `libtest.so` si está en el directorio actual o en uno de los directorios de búsqueda de bibliotecas estándar del sistema:

```
% gcc -o aplicación aplicación.o -L. -ltest
```

Supongamos que ambos `libtest.a` y `libtest.so` están disponibles. Luego, el vinculador debe elegir una de las bibliotecas y no la otra. El enlazador busca en cada directorio (primero los especificados con `-L` opciones, y luego las de los directorios estándar). Cuando el enlazador encuentra un directorio que contiene `libtest.a` o `libtest.so`, el enlazador detiene la búsqueda de directorios. Si solo una de las dos variantes está presente en el directorio, el vinculador elige esa variante. De lo contrario, el vinculador elige la versión de la biblioteca compartida, a menos que le indique explícitamente lo contrario. Puedes usar el `-static` opción de exigir archivos estáticos. Por ejemplo, la siguiente línea usará el `libtest.a` archivo, incluso si el

`libtest.so` La biblioteca compartida también está disponible:

```
% gcc -static -o aplicación aplicación.o -L. -ltest
```

los `ldd` El comando muestra las bibliotecas compartidas que están vinculadas a un ejecutable. Estas bibliotecas deben estar disponibles cuando se ejecuta el ejecutable. Tenga en cuenta que `ldd` enumerará una biblioteca adicional llamada `ld-linux.so`, que es parte del mecanismo de enlace dinámico de GNU / Linux.

Usando `LD_LIBRARY_PATH`

Cuando vincula un programa con una biblioteca compartida, el vinculador no coloca la ruta completa a la biblioteca compartida en el ejecutable resultante. En su lugar, coloca solo el nombre de la biblioteca compartida. Cuando el programa se ejecuta realmente, el sistema busca la biblioteca compartida y la carga. El sistema solo busca `/lib` y `/usr / lib`, por defecto. Si una biblioteca compartida que está vinculada a su programa se instala fuera de esos directorios, no se encontrará y el sistema se negará a ejecutar el programa.

Una solución a este problema es utilizar el `-Wl, -rpath` opción al vincular el programa. Suponga que usa esto:

```
% gcc -o aplicación aplicación.o -L. -ltest -Wl, -rpath, /usr / local / lib
```

Entonces cuando `aplicación` se ejecuta, el sistema buscará `/usr / local / lib` para las bibliotecas compartidas necesarias.

Otra solución a este problema es configurar el `LD_LIBRARY_PATH` variable de entorno al ejecutar el programa. Como el `SENDERO` Variable ambiental, `LD_LIBRARY_PATH` es un colon separado lista de directorios. Por ejemplo, si `LD_LIBRARY_PATH` es `/usr / local / lib: / opt / lib`, luego `/usr / local / lib` y `/opt / lib` se buscará antes que el estándar `/lib` y `/usr / lib` directorios. También debe tener en cuenta que si tiene `LD_LIBRARY_PATH`, el enlazador buscará los directorios dados allí además de los directorios dados con el `-L` opción cuando está construyendo un ejecutable. ^[4]

^[4] Es posible que vea una referencia a `LD_RUN_PATH` en alguna documentación en línea. No crea lo que lee; esta variable en realidad no hace nada bajo GNU / Linux.

2.3.3 Bibliotecas estándar

Incluso si no especificó ninguna biblioteca cuando vinculó su programa, es casi seguro que use una biblioteca compartida. Eso es porque GCC se vincula automáticamente en la biblioteca C estándar, `libc`, para ti. Las funciones matemáticas estándar de la biblioteca C no están incluidas en `libc`; en cambio, están en una biblioteca separada, `libm`, que debe especificar explícitamente. Por ejemplo, para compilar y vincular un programa `compute.c` que utiliza funciones trigonométricas como `pecado` y `porque`, debes invocar este código:

```
% gcc -o compute compute.c -lm
```

Si escribe un programa en C ++ y lo vincula utilizando el `c ++` o `g ++` comandos, también obtendrá la biblioteca estándar de C ++, `libstdc ++`, automáticamente.

2.3.4 Dependencias de la biblioteca

Una biblioteca a menudo dependerá de otra biblioteca. Por ejemplo, muchos sistemas GNU / Linux incluyen `libtiff`, una biblioteca que contiene funciones para leer y escribir archivos de imagen en formato TIFF. Esta biblioteca, a su vez, utiliza las bibliotecas `libjpeg` (Rutinas de imagen JPEG) y `libz` (rutinas de compresión).

[Listado 2.9](#) muestra un programa muy pequeño que utiliza `libtiff` para abrir un archivo de imagen TIFF.

Listado 2.9 (tifftest.c) Usando libtiff

```
# incluye <stdio.h>
# incluye <tiffio.h>
int main (int argc, char ** argv) {

    TIFF * tiff;
    tiff = TIFFOpen (argv [1], "r"); TIFFClose
    (tiff);
    return 0;
}
```

Guarde este archivo fuente como `tifftest.c`. Para compilar este programa y vincularlo con `libtiff`, especificar `-ltiff` en su línea de enlace:

```
% gcc -o tifftest tifftest.c -ltiff
```

De forma predeterminada, esto recogerá la versión de biblioteca compartida de `libtiff`, encontrado en `/usr/lib/libtiff.so`. Porque `libtiff` usos `libjpeg` y `libz`, las versiones de biblioteca compartida de estos dos también se dibujan (una biblioteca compartida puede apuntar a otras bibliotecas compartidas de las que depende). Para verificar esto, use el `ldd` mando:

```
% ldd tifftest
```

```
libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000) libc.so.6 => /lib/libc.so.6
(0x40060000) libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
libz.so.1 => /usr/lib/libz.so.1 (0x40174000) /lib/ld-linux.so.2 => /lib/ld-
linux.so.2 (0x40000000)
```

Las bibliotecas estáticas, por otro lado, no pueden apuntar a otras bibliotecas. Si decide enlazar con la versión estática de `libtiff` especificando `-estático` en su línea de comando, encontrará símbolos sin resolver:

```
% gcc -static -o tifftest tifftest.c -ltiff /usr/bin/../lib/libtiff.a(tif_jpeg.o): En función 'TIFFjpeg_error_exit':
tif_jpeg.o (.text + 0x2a): referencia indefinida a 'jpeg_abort' /usr/bin/../lib/libtiff.a(tif_jpeg.o): En función
'TIFFjpeg_create_compress': tif_jpeg.o (.text + 0x8d): referencia indefinida a 'jpeg_std_error' tif_textjpeg.o
(. + 0xcf): referencia indefinida a 'jpeg_CreateCompress'
```

...

Para vincular este programa de forma estática, debe especificar las otras dos bibliotecas usted mismo:

```
% gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz
```

Ocasionalmente, dos bibliotecas serán mutuamente dependientes. En otras palabras, el primer archivo hará referencia a los símbolos definidos en el segundo archivo y viceversa. Esta situación generalmente surge de un diseño deficiente, pero ocasionalmente surge. En este caso, puede proporcionar una única biblioteca varias veces en la línea de comandos. El vinculador buscará en la biblioteca cada vez que ocurra. Por ejemplo, esta línea provocará `libfoo.a` para ser buscado varias veces:

```
% gcc -o app app.o -lfoo -lbar -lfoo
```

Entonces, incluso si `libfoo.a` símbolos de referencias en `libbar.a`, y viceversa, el programa se vinculará correctamente.

2.3.5 Pros y contras

Ahora que sabe todo sobre archivos estáticos y bibliotecas compartidas, probablemente se esté preguntando cuál usar. Hay algunas consideraciones importantes a tener en cuenta.

Una de las principales ventajas de una biblioteca compartida es que ahorra espacio en el sistema donde está instalado el programa. Si está instalando 10 programas y todos utilizan la misma biblioteca compartida, ahorrará mucho espacio al usar una biblioteca compartida. Si usó un archivo estático en su lugar, el archivo se incluye en los 10 programas. Por lo tanto, el uso de bibliotecas compartidas ahorra espacio en disco. También reduce los tiempos de descarga si su programa se descarga de la Web.

Una ventaja relacionada con las bibliotecas compartidas es que los usuarios pueden actualizar las bibliotecas sin actualizar todos los programas que dependen de ellas. Por ejemplo, suponga que produce una biblioteca compartida que administra conexiones HTTP. Muchos programas pueden depender de esta biblioteca. Si encuentra un error en esta biblioteca, puede actualizar la biblioteca. Instantáneamente, todos los programas que dependen de la biblioteca serán arreglados; no tiene que volver a vincular todos los programas de la misma manera que lo hace con un archivo estático.

Esas ventajas pueden hacerle pensar que siempre debe usar bibliotecas compartidas. Sin embargo, existen razones sustanciales para utilizar archivos estáticos en su lugar. El hecho de que una actualización a una biblioteca compartida afecte a todos los programas que dependen de ella puede ser una desventaja. Por ejemplo, si está desarrollando software de misión crítica, es posible que prefiera vincular a un archivo estático para que una actualización a compartido

las bibliotecas del sistema no afectarán a su programa. (De lo contrario, los usuarios podrían actualizar la biblioteca compartida, rompiendo así su programa, y luego llamar a su línea de atención al cliente, culpándolo a usted).

Si no podrá instalar sus bibliotecas en `/lib` o `/usr / lib`, definitivamente deberías pensarlo dos veces antes de usar una biblioteca compartida. (No podrá instalar sus bibliotecas en esos directorios si espera que los usuarios instalen su software sin privilegios de administrador). `-Wl, -rpath` El truco no funcionará si no sabe dónde van a terminar las bibliotecas. Y pedirle a sus usuarios que establezcan `LD_LIBRARY_PATH` significa un paso adicional para ellos. Debido a que cada usuario tiene que hacer esto individualmente, esta es una carga adicional sustancial.

Tendrá que sopesar estas ventajas y desventajas para cada programa que distribuya.

2.3.6 Carga y descarga dinámica

A veces, es posible que desee cargar algún código en tiempo de ejecución sin vincularlo explícitamente en ese código. Por ejemplo, considere una aplicación que admita módulos "plug-in", como un navegador web. El navegador permite a los desarrolladores de terceros crear complementos para proporcionar funcionalidad adicional. Los desarrolladores externos crean bibliotecas compartidas y las colocan en una ubicación conocida. A continuación, el navegador web carga automáticamente el código en estas bibliotecas.

Esta funcionalidad está disponible en Linux mediante el `abrir` función. Podrías abrir una biblioteca compartida llamada `libtest.so` llamando `abrir` como esto:

```
dlopen ("libtest.so", RTLD_LAZY)
```

(El segundo parámetro es una bandera que indica cómo enlazar símbolos en la biblioteca compartida. Puede consultar las páginas de manual en línea para `abrir` si quieres más información, pero `RTLD_LAZY` suele ser la configuración que desea.) Para utilizar las funciones de carga dinámica, incluya el `<dlfcn.h>` archivo de encabezado y enlace con el `-ldl` opción para recoger la biblioteca `libdl`.

El valor de retorno de esta función es un `vacío *` que se utiliza como identificador de la biblioteca compartida. Puede pasar este valor al `dlsym` función para obtener la dirección de una función que se ha cargado con la biblioteca compartida. Por ejemplo, si `libtest.so` define una función llamada `mi_función`, podrías llamarlo así:

```
void * handle = dlopen ("libtest.so", RTLD_LAZY); void (* prueba)
() = dlsym (identificador, "mi_función"); (*prueba)();
```

```
dlclose (mango);
```

los `dlsym` La llamada al sistema también se puede utilizar para obtener un puntero a una variable estática en la biblioteca compartida.

Ambos `abrir` y `dlsym` regreso `NULO` si no lo consiguen. En ese caso, puede llamar `dlerror` (sin parámetros) para obtener un mensaje de error legible por humanos que describe el problema.

los `cerrar` La función descarga la biblioteca compartida. Técnicamente, `abrir` realmente carga la biblioteca solo si aún no está cargada. Si la biblioteca ya se ha cargado, `abrir` simplemente incrementa el recuento de referencias de la biblioteca. Similar, `cerrar` disminuye el recuento de referencias y luego descarga la biblioteca solo si el recuento de referencias ha llegado a cero.

Si está escribiendo el código en su biblioteca compartida en C ++, probablemente querrá declarar esas funciones y variables a las que planea acceder en otro lugar con el `externo "C"` especificador de vinculación. Por ejemplo, si la función C ++ `mi_función` está en una biblioteca compartida y desea acceder a ella con `dlsym`, deberías declararlo así:

```
extern "C" void foo ();
```

Esto evita que el compilador de C ++ altere el nombre de la función, lo que cambiaría el nombre de la función de `foo` a un nombre diferente y de aspecto divertido que codifica información adicional sobre la función. El compilador de C++ no modificará los nombres; usará el nombre que le dé a su función o variable.