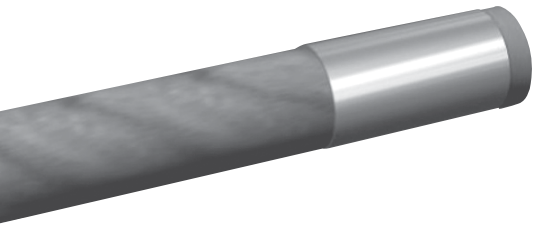
Capítulo 33

Flujos y archivos en C++



Contenido

Introducción

Hasta este momento se han realizado las operaciones básicas de entrada y salida. La operación de introducir (*leer*) datos en el sistema se denomina **lectura** y la generación de datos del sistema se denomina **escritura**. La lectura de datos se realiza desde el teclado e incluso desde la unidad de disco, y la escritura de datos se realiza en el monitor y en la impresora del sistema.

Al igual que sucede en C ANSI, las funciones de entrada/salida no están definidas en el propio lenguaje C++, sino que están incorporadas en cada compilador de C++ bajo la forma de *biblioteca de ejecución.* En C existe la biblioteca `stdio.h` estandarizada por ANSI; en C++ la *biblioteca* corres-

pondiente es *iostream,* aunque en este caso todavía no está estandarizada. En consecuencia, el lector puede recurrir en las operaciones de E/S (entrada/salida) a cualquiera de las dos bibliotecas, aunque la biblioteca *iostream* se distingue positivamente de *stdio.h* en que la entrada/salida se realiza por **flujos** (*streams*). Este método además de proporcionar la funcionalidad de C, es flexible y eficiente mediante la gestión de clases, que permite sobrecargar funciones y operadores, lo que hará que sus clases puedan ser manipuladas como si fueran tipos predefinidos.

La gran ventaja de la biblioteca *iostream* es que se pueden manipular las operaciones E/S sin necesidad de conocer los conceptos típicos de orientación a objetos, tales como clases, herencia, funciones virtuales, etcétera.

En este capítulo aprenderá a utilizar las características típicas de E/S de C++ y obtener el máximo rendimiento de las mismas.

La biblioteca de flujos `iostream` de C++ contiene la clase **ios**. Esta clase declara los identificadores que establecen el modo de flujos de archivos. La biblioteca tiene las clases `ifstream`, `ofstream` y `fstream`, que admiten flujos de archivos de entrada, de salida y de entrada/salida. Este capítulo examina las funciones de la biblioteca de flujos que admite E/S de archivos de texto, E/S de archivos binarios y E/S de archivos de acceso aleatorio.

Los tipos de clases proporcionan el siguiente soporte de archivos:

- `ifstream`, derivada de `istream`, conecta un archivo al programa para entrada.
- `of stream`, derivada de `ostream`, conecta un archivo al programa para salida.
- `fstream`, derivada de `iostream`, conecta un archivo al programa para entrada y salida.

Para usar el componente de flujos de archivos de la biblioteca `iostream` se debe incluir su archivo asociado: `#include <fstream>`.

Conceptos clave

- Acceso aleatorio
- Apertura de un archivo
- Archivo de cabecera
- Archivo de texto
- Archivos binarios
- Biblioteca
- Biblioteca de clases
- Bit de estado
- Entrada/Salida
- Final del archivo
- Flujo (*stream*)
- Flujo binario
- Flujo de texto
- Indicador de estado
- Manipulador
- Operador de extracción
- Operador de inserción

Flujos (*streams*)

Un *flujo* (*stream*) es una abstracción que se refiere a un *flujo* o *corriente* de datos entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o tubería *(pipe*) por la que circulen los datos. Estas conexiones se realizan mediante operadores (`<< y >>`) sobrecargados y funciones de E/S.

Un **flujo** es una abstracción desarrollada por Bjarne Stroustrup, el diseñador de C++, sobre la idea original de C y UNIX. Stroustrup se imaginó un flujo (corriente) de caracteres fluyendo desde el teclado a un programa, al igual que un flujo de agua fluye de un sitio a otro. Schwarz utilizó esta idea para crear la clase `istream`, una clase que representa un flujo de caracteres desde un dispositivo de entrada arbitrario a un programa en ejecución.

En esencia, un **flujo** es una abstracción que se refiere a una interfaz común a diferentes dispositivos de entrada y salida de una computadora. Existen dos formas de flujo: texto y binario. Los ***flujos de texto*** se utilizan con caracteres ASCII, mientras que los ***flujos binarios*** se pueden utilizar con cualquier tipo de dato. Los sinónimos *extraer* u *obtener* se utilizan generalmente para referirse a la entrada de datos de un dispositivo e *inserción* o *colocación* (poner) cuando se refieren a la salida de datos a un dispositivo.

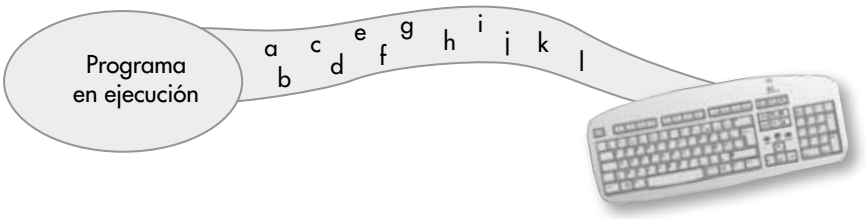


Figura 33.1 Simulación de un flujo.

Flujos de texto

Un *flujo de texto* es una secuencia de caracteres. En un flujo de texto, pueden ocurrir ciertas conversiones de caracteres si son requeridas por el entorno del sistema. Por ejemplo, un carácter de nueva línea puede convertirse en un par de caracteres "retorno de carro/salto de línea". Por esta razón, puede suceder que no se establezca una relación de uno a uno entre los caracteres que se escriben o se leen y los que aparecen en el dispositivo externo. De igual forma, como consecuencia de las posibles conversiones puede suceder que el número de caracteres escritos o leídos no coincida con el del dispositivo externo.

Flujos binarios

Un *flujo binario* es una secuencia de *bytes* que tiene una correspondencia uno a uno con los del dispositivo externo. Es decir, no se producen conversiones de caracteres. En este caso, la cantidad de *bytes* escritos o leídos coincide con la del dispositivo externo. Sin embargo, se puede añadir un número de *bytes* nulos definidos en la implementación, a un flujo binario. Estos *bytes* nulos se suelen utilizar, por ejemplo, para ajustar la información de tal forma que se complete un sector de un disco.

A recordar

Un **flujo** (*stream*) es una secuencia de caracteres.

Hasta este momento del libro, la entrada y la salida en C se ha implementado utilizando el flujo de entrada estándar (denominado `cin`) y el flujo de salida estándar (denominado `cout`). Todo programa C++ tiene estos flujos disponibles automáticamente siempre que se incluya el *archivo de cabecera `iostream.h`*; `cin` y `cout` son objetos de tipos `istream` y `ostream` respectivamente. Los tipos definidos por el usuario `istream` y `ostream` están definidos en las clases de la biblioteca `iostream` y `cin/cout` se declaran como objetos de esos tipos. Normalmente, `cin` se conecta al teclado. La lectura de caracteres desde el objeto `cin` del flujo de entrada estándar es equivalente a la lectura del teclado; la escritura de caracteres al `cout` del flujo de salida estándar es equivalente a visualizar estos caracteres en la pantalla.

Todas las características de flujos *entrada/salida* se relacionan con conversiones de representaciones internas de datos (variables u objetos) a un flujo de caracteres (para salida) y conversión de flujos de caracteres a un formato interno correcto (para entrada). La figura 33.2 muestra esta conversión para los flujos `cin` y `cout`. Mediante el operador de salida, `<<`, se convierten los datos que aparecen en el operando a un flujo de caracteres y "se insertan" en el flujo de salida (`cout`). Al contrario, el operador de entrada, `>>`, especifica "la extracción del" flujo de entrada de un flujo de caracteres. Estos caracteres se convierten entonces al formato interno apropiado y se almacenan en las posiciones de almacenamiento especificados.

Las clases de flujo de E/S

El archivo de cabecera `<iostream>` declara tres clases para los flujos de entradas y salida estándar. La clase `istream` es para entrada de datos desde un flujo de entrada, la clase `ostream` es para salida de datos a un flujo de salida, y la clase `iostream` es para entrada de datos dentro de un flujo. Por otra parte, estas clases declaran también los cuatro objetos ya conocidos (tabla 33.1).

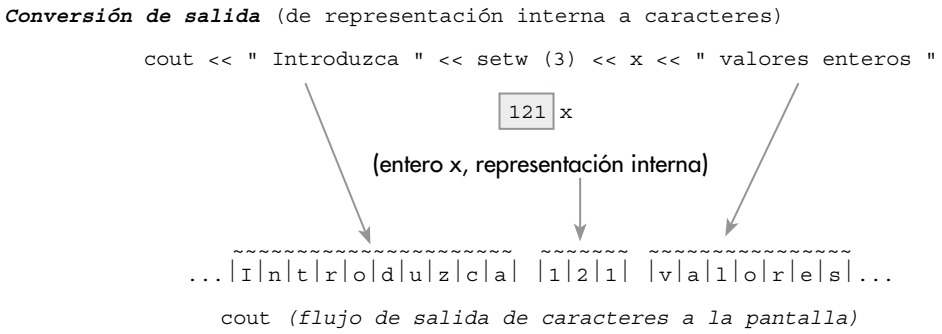


Figura 33.2 Conversión de datos a/desde flujos.

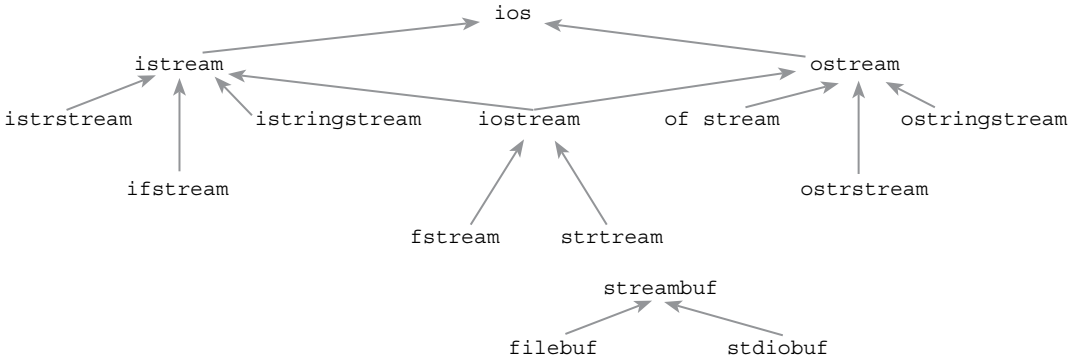


Figura 33.3 Biblioteca de clases de E/S.

Las clases que se derivan de la clase base `ios` se utilizan para procesamiento de flujos de alto nivel, mientras que las clases que se derivan de la clase base `streambuf` se utilizan para procesamiento de bajo nivel.

La clase `iostream` es la que se utiliza normalmente en operaciones ordinarias de E/S. Esta clase es una subclase de las clases `istream` y `ostream`, que a su vez son clases derivadas (subclases) de la clase base `ios`. Las tres clases que incluyen la palabra `fstream` en su nombre se utilizan para

Tabla 33.1 Objetos de `<iostream.h>`.

Objeto de flujo	Función
`cin`	Un objeto de la clase `istream` conectado a la entrada estándar.
`cout`	Un objeto de la clase `ostream` conectado a la salida estándar.
`cerr`	Un objeto de la clase `ostream` conectado al error estándar, para salida sin búfer.
`clog`	Un objeto de la clase `ostream` conectado al error estándar, con salida a través de búfer.

tratamiento de archivos. Por último, la clase `stdiobuf` se utiliza para combinar E/S de flujos C++ con las funciones antiguas de E/S estilo C.

Archivos de cabecera

Existen tres archivos de cabecera importantes para clases de flujos de E/S. El archivo de cabecera `<iostream>` declara las clases `istream`, `ostream` e `iostream` para las operaciones de E/S de flujos de entrada y salida estándar. Declara también los objetos `cout`, `cin`, `cerr` y `clog` que se utilizan en la mayoría de los programas C++.

El archivo de cabecera `<fstream>` declara las clases `ifstream`, `of stream` y `fstream` para operaciones de E/S a archivos de disco. Por último, el archivo de cabecera `<strstream>` declara las clases `istrstream`, `ostrstream` y `strstream` para formateado de datos con búfers de caracteres.

```
#include<iostream>
```

en cada programa que utilice E/S se ha de utilizar la directiva. El archivo de cabecera `iostream` incluye las definiciones de la biblioteca de E/S.

La biblioteca de clases `iostream`

La biblioteca `iostream` se basa en el concepto de *flujos*; incorpora la ventaja de las potentes características orientadas a objetos de C++.

La biblioteca de E/S de flujos se construye a base de una jerarquía de clases que se declaran en diversos archivos de cabecera. La *biblioteca de clases* tiene dos familias paralelas de clases: las derivadas de `streambuf` y las derivadas de `ios`.

La clase `streambuf`

La clase `streambuf` proporciona una interfaz a dispositivos físicos; proporciona métodos fundamentales para realizar operaciones con *buffer* y manejo de flujos cuando las condiciones de *`formateado`* no son muy exigentes.

Jerarquía de clases `ios`

La jerarquía de clases `ios` gestiona todas las operaciones de E/S y proporciona la interfaz de bajo nivel al programador. La clase `ios` contiene un puntero a `streambuf`.

Para acceder a la biblioteca `iostream` se deben incluir archivos de cabecera específicos; uno de ellos ya ha sido utilizado por el lector, *`iostream`*, pero existen otros archivos de cabecera, como se verá en esta misma sección.

La clase `istream` (*`input stream`*) proporciona las operaciones de lectura de datos, mientras que la clase `ostream` (*`output stream`*) implementa las operaciones de escritura de datos. La clase `iostream` (*`input-output stream`*) se deriva simultáneamente de `istream` y `ostream`, y proporciona operaciones bidireccionales de entrada/salida (es un ejemplo de *herencia múltiple*).

Las clases `istrstream` y `ostrstream` se utilizan cuando se desea manejar arrays de caracteres y flujos, mientras que el otro conjunto de clases `istringstream` y `ostringstream`, se utilizan cuando se conectan flujos con objetos de la clase estándar `string`. La figura 33.4 muestra el modo en el que se relacionan unas clases con otras.

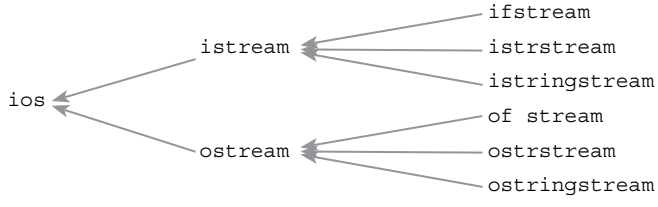


Figura 33.4 Clases derivadas de `ios`.

Las clases ios, istream, ostream y los objetos de flujos predeclarados (cin, cout, cerr y clog) se definen en el archivo iostream, el cual debe, por consiguiente, ser incluido en el programa. Las clases ifstream y of stream se definen en el archivo de cabecera fstream. El archivo de cabecera strstream contiene las definiciones de las clases istrstream y ostrstream y por último las clases istringstream y ostringstream están declaradas en el archivo de cabecera sstream.

Flujos estándar

La biblioteca iostream define cuatro flujos estándar (objetos de flujo predefinidos): cin, cout, cerr y clog (tabla 33.1). Estos tipos se declaran siempre automáticamente, de modo que no precisan declaración previa.

- El flujo cin, definido por la clase istream, está conectado al periférico de entrada estándar (el *teclado*, representado por el archivo stdin), aunque en algunos sistemas operativos podría ser redirigido (MS-DOS, Windows, Linux y UNIX).
- El flujo cout, definido por la clase ostream, está conectado al periférico de salida estándar (la *pantalla*, representado por el archivo stdout).
- El flujo cerr, definido por la clase ostream, está conectado al periférico de error estándar (la pantalla, representado por el archivo stdout). Este flujo no es a través de *buffer*.
- El flujo clog, definido por la clase ostream, está conectado igualmente al periférico de error estándar (la pantalla, representado por el archivo stdout). Al contrario que cerr, el flujo clog se realiza a través de *buffer*.

A recordar

Cualquier objeto creado de la clase ios o cualquiera de sus clases derivadas se conoce generalmente como un *objeto flujo*.

La ventaja de cerr sobre clog es que los *buffers* de salida se limpian (*vacían*) cada vez que cerr se utiliza, de modo que la salida está disponible más rápidamente en el dispositivo externo (que de manera predeterminada es la pantalla de video). Sin embargo, en grandes cantidades de mensajes, la versión clog a través de *buffer* es más eficiente.

Entradas/salidas en archivos

Las tres clases siguientes permiten efectuar entradas/salidas en archivos:

- ifstream, clase derivada de istream; se utiliza para gestionar la lectura de un archivo. Cuando se crea un objeto ifstream y se especifican parámetros, se abre un archivo.
- ofstream, clase derivada de ostream; gestiona la escritura en un archivo. Los objetos ofstream se utilizan para hacer operaciones de salida de archivos. Se declara un objeto ofstream si se piensa escribir un archivo de disco. Si se proporciona un nombre de archivo cuando se declara un objeto ofstream, se abre el archivo. Se puede especificar que el archivo se cree en modo binario o en modo texto. Si un objeto de of stream está ya declarado, se puede utilizar la función miembro open ( ) para abrir el archivo. Por otra parte, se dispone de la función miembro close ( ), que sirve para cerrar el archivo.
- fstream, clase derivada de iostream; permite leer y escribir en un archivo. Los objetos fstream se utilizan cuando se desea realizar en forma simultánea operaciones de lectura y escritura en el mismo archivo.

Las definiciones de estas clases se encuentran en el archivo de cabecera fstream.

Entradas/salidas en un búfer de memoria

Existen dos clases específicas destinadas a las entradas/salidas en un búfer en memoria:

- istrstream, clase derivada de istream, permite leer caracteres a partir de una zona de memoria, que sirve de flujo de entrada.
- ostrstream, clase derivada de ostream, permite escribir caracteres en una zona de memoria, que sirve de flujo de salida.

El archivo de cabecera `strstream` contiene las definiciones de las `clases istrstream` y `ostrstream`.

Archivos de cabecera

Cualquier programa que utilice la biblioteca `iostream` debe incluir el archivo de cabecera y, eventualmente, otros archivos de cabecera suplementarios: `<ios>`, `<istream>`, `<ostream>`, `<ifstream>`, `<ofstream>`, `<fstream>`, `<strstream>`, `<iomanip>`.

El archivo de cabecera `i` declara clases de bajo nivel e identificadores. Los archivos `istream.h` y `ostream.h` admiten las entradas y salidas básicas de los flujos. El archivo `iostream. h` combina las operaciones de las clases en los dos archivos de cabecera anteriores. Para realizar entradas/salidas en archivos, se deben incluir los archivos de cabecera `fstream` e `iostream`. El archivo `iomanip` permitirá *formatear* y organizar la salida de datos. La inclusión del archivo de cabecera `strstream` permite el acceso a las funciones de la biblioteca `iostream`, que permiten efectuar las entradas/salidas en memoria.

Entrada/salida de caracteres y flujos

C++ visualiza todas las entradas y salidas como flujos de caracteres. Si su programa obtiene la entrada del teclado, un archivo de disco, un módem o un ratón, C++ ve sólo un flujo de caracteres. C++ no conoce cuál es el tipo de dispositivo que le proporciona la entrada.

Estas operaciones de E/S de flujos significan que se utilizan las mismas funciones para obtener la entrada del teclado como del módem. Se pueden utilizar las mismas funciones para escribir en un archivo de disco, una impresora o una pantalla. Naturalmente, se necesita algún medio para encaminar el flujo de entrada o salida al dispositivo adecuado.

El flujo de datos irá de un dispositivo de entrada (teclado) al programa C++ y de un programa C++ al dispositivo de salida (la pantalla o la impresora).

Clases `istream` y `ostream`

Las clases `istream` y `ostream` se derivan de la clase base `ios`.

```
class istream: virtual public ios { //  ... };
class ostream: virtual public ios { //  ... };
```

Clase `istream`

La clase `istream` permite definir un flujo de entrada y admite métodos para entrada *formateada* y *no formateada.* El *operador de extracción* `>>` está sobrecargado para todos los tipos de datos integrales de C++, haciendo posible operaciones de entrada de alto nivel.

Declaración

```
class istream: virtual public ios   // iostream
{
  // ...
};
```

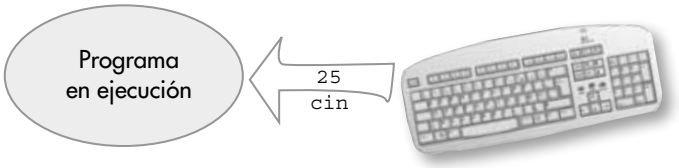
Un *objeto de flujo* es una instancia de una subclase de la clase `ios`. El operador de entrada `>>` se aplica a un objeto `istream` y a un objeto variable.

```
objeto_istream >> objeto_variable
```

y trata de extraer una secuencia de caracteres correspondientes a un valor del tipo de `objeto_variable` de `objeto_istream`. Si no hay caracteres, se bloquea la ejecución precedente hasta que se introducen caracteres. Como ejemplo, supongamos que `cin` está inicialmente vacío y se ejecutan las sentencias siguientes:

```
int edad;
cin >> edad;
```

Si el usuario introduce 25, los caracteres 2 y 5 se introducen en el objeto cin de istream. Se leen los caracteres 2 y 5 de iostream, los convierte al valor entero 25 y almacena este valor en su operando derecho edad.



Indicadores de estado

¿Qué sucederá si el usuario introduce valores no apropiados? Por ejemplo, en el caso anterior, en lugar de escribir 25 se comete un error y se teclea t5. Entonces el flujo cin contendrá los caracteres t y 5. Es decir, el operador >> trata de extraer un entero y encuentra el carácter t.

La clase istream tiene un atributo que es su **estado** o **condición**. El estado se representa mediante *indicadores* o *banderas* (*flags*) que representan la condición o estado de un flujo (tabla 32.2). Las tres condiciones básicas son: good (el estado es bueno); bad (hay algo incorrecto en el flujo); fail (la última operación del flujo no tuvo éxito). La clase istream contiene una variable booleana llamada **bandera** para cada uno de estos estados, con el indicador good inicializado a true (verdadero) y los indicadores bad y fail inicializados a falso.

En el ejemplo anterior, al encontrar la letra t cuando se intentaba leer un entero, se pone el indicador good del flujo a falso, el indicador bad a true (verdadero) y fail también verdadera, y viceversa. Para cada uno de estos indicadores, la clase istream proporciona una función miembro boolean que tiene el mismo nombre que su indicador, que informa del valor de ese indicador.

A recordar

El operador >> se suele denominar *operador de extracción* debido a que su comportamiento es extraer valores de una clase istream.

Variables de estado de ios

La clase ios tiene unas variables de estado que se especifican en la definición enum.

```
class ios {
public:
  enum {                    // valor del indicador de estado de error
    goodbit =  0,           // todos correctos
    eofbit  = 01,           // fin de archivo
    failbit = 02,           // última operación fallada
    badbit  = 04            // operación no válida
  };
  // otros miembros incluidos aquí
};
```

Los indicadores de formato de flujo sólo se pueden cambiar explícitamente y sólo mediante las funciones de acceso de ios. En contraste, variables de estado de flujo se cambian implícitamente, como resultado de operaciones de E/S. Por ejemplo, cuando un usuario introduce Control-D (o Control-Z en computadoras DOS y VAX) para indicar el *final del archivo*; el *indicador de estado eof* de cin se pone a 1, y se dice que el flujo está en un estado *eof*.

Tabla 33.2 Indicadores de estado.

Llamada a función	Devuelve *true* si y sólo si
cin.good ( )	Todo está correcto en istream
cin.bad ( )	Algo está mal en el istream
cin.fail ( )	No se puede completar la última operación

Se puede acceder individualmente a las cuatro *variables de estado* (`goodbit`, `failbit`, `eofbit` y `badbit`) por sus funciones de acceso [`good ( )`, `fail ( )` y `bad ( )`]. Puede accederse también a todas ellas colectivamente mediante la función `rdstate ( )`.

Ejemplo 33.1

```
main( )
{
  cout << "cin.rdstate( ) = " << cin.rdstate << endl;
  int n;
  cin >> n;
  cout << "cin.rdstate( ) = " << cin.rdstate() << endl;
}
```

Ejecución

```
cin.rdstate( ) = 0
22
cin.rdstate( ) = 0
```

Segunda ejecución

```
cin.rdstate( ) = 0
^D control-D
cin.rdstate( ) = 3
```

En la segunda ejecución, el usuario pulsó ***Control-D*** para señalar el final del archivo. Esta acción fija a `eofbit` y `failbit` de `cin`, que tienen valores numéricos 1 y 2, toman el valor total 3 de la variable de estado `_state`.

***La función* `clear( )`**

El uso de **Control-D** (o **Control-Z)** para terminar la entrada es sencillo y adecuado. Pulsando esta secuencia de teclas, se fija el valor de `eofbit` en el flujo de entrada. Pero a continuación, si se desea utilizarlo de nuevo en el mismo programa, se ha de borrar (limpiar) primero. Esta acción se realiza con la función `clear( )`.

Ejemplo 33.2

```
int main( )
{
               int n, suma = 0;
               while (cin >> n)
                 suma += n;
  cout << "La suma parcial es " << suma << endl;
  cin.clear();
               while (cin >> n)
                 suma += n;
  cout << "La suma total es" << suma << endl;
  return 0;
}
```

Ejecución

```
30   20   50   ^D
La suma parcial es 100
40   80
La suma total es 220
```

Clase ostream

La clase ostream permite a un usuario definir un flujo de salida y admite métodos para salidas *formateadas* y *no formateadas*. El *operador de inserción* se sobrecarga para todos los tipos de datos integrales. Las clases of stream, ostrstream, constream, iostream y ostream_withassign se derivan todas de ostream.

Al igual que istream, la clase ostream se deriva virtualmente de la clase ios para evitar declaraciones múltiples cuando se declara iostream.

```
class ostream: virtual public ios   // iostream
{
  // ...
};
```

La noción abstracta de un flujo se puede utilizar para ocultar estos detalles de bajo nivel del programador, y de este modo la biblioteca iostream proporciona la clase ostream para representar el "flujo" de caracteres de un programa en ejecución a un dispositivo de salida arbitrario (véase la figura 33.5).

Después de crear una clase ostream, se definen dos objetos ostream de la biblioteca iostream de modo que cualquier programa que incluye el archivo de cabecera de la biblioteca tiene dos flujos de salida del programa a cualquier dispositivo que el usuario esté utilizando para salida: una ventana, una terminal, etcétera.

1. cout, un ostream para visualizar salida normal.
2. cerr, un ostream para visualizar mensajes de error o de diagnóstico.

El mecanismo assert( ) normalmente escribe sus mensajes de diagnóstico a cerr, no a cout.

Figura 33.5 Simulación de flujo ostream.

El operador <<

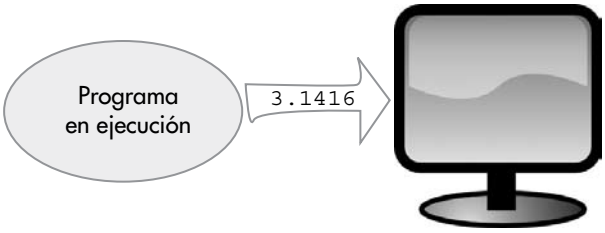
Cuando el operador << se aplica a un objeto ostream y una expresión

```
objeto_ostream << expresión
```

se evaluará la expresión y se inserta la secuencia de caracteres correspondientes a ese valor en el objeto ostream. Por consiguiente, en el caso de:

```
const double PI = 3.1416;
cout << PI;
```

la función << convierte el valor double, 3.1416, en los caracteres correspondientes 3, ., 1, 4, 1 y 6 y los inserta uno a uno en cout:



Los caracteres permanecen realmente en el `ostream`, sin aparecer en la pantalla, hasta que se *limpia* (*fluye*) el `ostream`, que, como su nombre sugiere, vacía el flujo en la pantalla.

Ejemplo 33.3

```
double f(double x, double y)
{
  cout << "Introducción de f";
  .
  .
  .
  cout << "salida de f";
  return z;
}
```

La salida `"Introducción de f"` puede no aparecer en nuestra pantalla cuando se espera ya que el `ostream` no se ha limpiado.

Manipulador de salida

Un medio común para limpiar un `ostream` es utilizar un **manipulador de salida**, un identificador que afecta al propio `ostream` cuando se utiliza en una sentencia de salida, en lugar de generar simplemente un valor que aparece en la pantalla. El *manipulador* más utilizado para limpiar un flujo de salida es el **manipulador** `endl`:

```
double f(double x, double y)
{
cout << "Introducción de f" << endl;
  ...
  return z;
}
```

Este manipulador inserta un carácter de nueva línea `('\n')` en el flujo `ostream` y lo limpia, terminando una línea de salida. Otro manipulador es `flush` que limpia el flujo `ostream` sin insertar nada:

```
cout << "Salida de f" << flush;
```

Salida a la pantalla y a la impresora

Cuando se ejecutan los programas C++ normalmente se deseará generar información en uno de los dos dispositivos *hardware* típicos: un monitor o una impresora. De hecho, habrá ocasiones en que se necesitará generar información en ambos dispositivos durante la ejecución de un programa.

Se puede escribir información en el dispositivo de salida utilizando el objeto `cout`, que está definido en el archivo de cabecera `iostream`. Los objetos son el núcleo de la programación orientada a objetos y su uso correcto en C++ potenciará los programas.

El método más común de dirigir la salida a la pantalla es utilizar el objeto `cout`. El flujo de salida que se pasa al objeto `cout` se dirige al flujo de salida estándar. Mediante el operador de inserción (`<<`) se ponen datos en el flujo de salida.

```
cout << "Hola mundo, C++";
```

El operador de inserción se define para todos los tipos de datos básicos: `char`, `unsigned char`, `signed char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, `void*` y `char*` (un puntero a una cadena). El operador de inserción convier-

te los datos a la derecha de << a una cadena de caracteres (char), tipo esperado por el objeto cout. Por ejemplo, el siguiente valor entero, val_ent, se convierte a la cadena 47 y se pasa al objeto cout:

```
int val_ent = 47;
cout << val _ent;
```

En el caso anterior se visualiza 47. La variable val_ent es una expresión, por lo que podría también ser válida la sentencia:

```
int i, j;
cout << i+j;
```

Operadores de inserción en cascada

El operador de inserción se puede poner en cascada, de modo que pueden aparecer varios elementos en una sola sentencia C++. Así, la sentencia

```
cout << 1 << 2 << 3 << 4;
```

generará una salida tal como

```
1 2 3 4
```

Si se desea escribir información de caracteres, se debe encerrar la información de salida entre comillas. La sentencia

```
cout << "Hola, programador de C++";
```

genera la salida

```
Hola, programador de C++
```

Los operadores en cascada pueden mezclar valores de caracteres y numéricos. Así, por ejemplo,

```
cout << "Total =" << Suma << endl;
```

visualizará

```
Total = 450
```

Suponiendo que el valor de la variable Suma es 450. El símbolo endl, como ya conoce el lector, hace que el flujo de salida avance a la siguiente línea. Se puede situar endl en cualquier parte del flujo, aunque suele situarse al final de la línea. La sentencia siguiente:

```
cout<< "Los resultados son los siguientes:" << endl;
cout<<  "Enero   "  <<  Total_Ene   <<  endl;
cout<<  "Febrero "  <<  Total _Feb  <<  endl;
cout<<  "Marzo   "  <<  Total_Mar   <<  endl;
```

produce la salida

```
Los resultados son los siguientes:
Enero   300
Febrero 425
Marzo   106
```

Desde el punto de vista práctico, cada operador de inserción envía un dato (una constante, una expresión o una variable) al flujo de salida, y se pueden concatenar datos de tipos diferentes en una única expresión cout .

```
cout << Voltaje << Corriente << Resistencia;
```

La sentencia precedente escribirá los valores almacenados en memoria en el mismo orden en que están escritos. El orden de salida será el mismo que el orden listado en la sentencia cout, pero sin ningún espacio entre los valores. Si desea espaciado entre los valores deberá insertar caracteres en blanco dentro de la sentencia cout .

Las funciones miembro `put ( )` y `write ( )`

Las clases definen datos y funciones miembro. Una función de una clase se llama *método* o *función miembro*. La clase `ostream` proporciona la función **`put ( )`** para insertar un único carácter en el flujo de salida y la función **`write ( )`**, para insertar una cadena en el flujo de salida. Ambas funciones devuelven un objeto `cout`.

Se puede escribir en un flujo de salida llamando a las funciones miembro **`put ( )`** o **`write ( )`**. Los formatos de estas funciones son:

```
dispositivo. put (valor_caracter);
dispositivo.write (valor _cadena, num);
```

El punto que separa *dispositivo* de la función **`put ( )`** es el operador de miembro `(.)`. *`valor_caracter`* puede ser una constante, expresión o variable carácter (`char`) y *`valor_cadena`* una cadena; *num* es un valor `int` utilizado para especificar el número de caracteres de la cadena a visualizar. El *`dispositivo`* puede ser cualquier dispositivo de salida estándar. Para escribir, por ejemplo, un carácter en su impresora, se abrirá `PRN` con `of stream`.

Así, las sentencias siguientes visualizan dos caracteres (`'Z'` y `'l'`) en el flujo de salida:

```
cout.put('Z');
char letra = 'l';
cout.put(letra);
```

Si se desea escribir un bloque de caracteres, se utiliza la función miembro `write`, como en estos ejemplos:

```
cout.write("Biblioteca", 3);     // Se visualiza Bib
cout.write("Día Nacional", 12) << "\n";
cout.put(65) << "Antonio Molina \n";
cout.put('H').write("ola", 4) << " mundo C++ \n";
```

Al ejecutarse las tres sentencias anteriores se visualiza

```
Día Nacional
Antonio Molina
Hola mundo C++
```

Obsérvese que la primera función **`put ( )`** contiene una constante de valor entero; el entero se interpreta como un código `ASCII` de la letra `A` que se visualiza.

La función **`write ( )`** visualiza tantos caracteres como se especifique en el segundo argumento. Si se especifica un número mayor que el número de caracteres de la cadena, la función visualiza cualquier cosa que resida en memoria a continuación de la cadena.

Impresión de la salida en una impresora

El envío de la salida de un programa a la impresora es fácil con la clase `ofstream`. El formato es:

```
ofstream dispositivo (nombre_dispositivo)
```

y su uso requiere el archivo de cabecera `fstream`.

Ejemplo 33.4

El siguiente programa solicita al usuario el nombre y los apellidos. A continuación, imprime el nombre completo y el apellido en la impresora.

```
// SALIMPRE.CPP
// Imprime un nombre en la impresora
```

```
#include <fstream>
using namespace std;

int main( )
{
  char nombre[20];
  char apellidos[30];

  cout << "¿Cuál es su nombre?";
  cin >> Nombre;
  cout << "¿Cuáles son sus apellidos?";
  cin >> Apellidos;

  // Enviar nombre y apellidos a la impresora

  ofstream impresora ("PRN");
  impresora << "Su nombre completo es: \n";
  impresora << apellidos << ", " << nombre << endl;

  return 0;
}
```

Lectura del teclado

Obtener información en un programa para su procesamiento se denomina *lectura.* En la mayoría de los sistemas actuales, la información se lee de una de las dos fuentes: de un teclado o de un archivo de disco. En esta sección, aprenderá cómo se lee la información procedente del teclado.

La sentencia C++ que se utiliza para la lectura de datos del teclado es `cin`. Al igual que `cout`, `cin` es un objeto predefinido en C++ y es parte del archivo de cabecera `iostream`.

Cuando se teclea, se genera un flujo de entrada. Al igual que con la salida, C++ utiliza un enfoque orientado a objetos para la entrada. El objeto `cin` extrae caracteres del flujo de entrada, lo convierte a cualquier tipo de dato diseñado en la sentencia de entrada y lo almacena en la posición de memoria deseada.

Se utiliza el operador de extracción `>>` para manejar la entrada de un flujo. El operador obtiene datos del flujo y lo sitúa en una variable.

Un ejemplo de una sentencia de entrada en C++ es:

```
int valor;
cin >> valor;
```

El operador de extracción se utiliza con el objeto `cin` para introducir datos desde el teclado. El operador `>>` es fácil de recordar, ya que sugiere un flujo de datos desde la izquierda a la derecha. El operador se denomina operador de extracción, ya que *extrae* datos desde el flujo de entrada.

Al igual que el operador de inserción, el operador de extracción se define para todos los tipos de datos básicos: `char`, `unsigned char`, `signed char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, cadenas y punteros. El operador de extracción convierte los datos desde el flujo de entrada al tipo de datos esperado por la variable que recibe el dato.

El operador de extracción se suele utilizar en unión con un operador de inserción y un mensaje de petición de datos o salutación:

```
int Edad;
cout << "Introduzca edad del alumno:";
cin >> Edad;
```

Al igual que el operador de inserción, el operador de extracción se puede poner en cascada, con un formato similar a:

```
cin >> variable1 >> variable2 >> ... >> variablen
```

Ejemplo 33.5

```
int Edad;
float Altura;
cout << "Introduzca Edad y Altura:";
cin >> Edad >> Altura;
```

Cuando se procesan elementos múltiples en la entrada, se debe teclear al menos un espacio en blanco entre los elementos de entrada. La entrada a las sentencias anteriores podría ser:

```
Introduzca Edad y Altura: 47 75
```

Ejemplo 33.6

Introduzca un valor entero desde el teclado y, a continuación, visualícelo.

```
// ENDATOS1.CPP - Introducir un número utilizando el objeto cin

#include <iostream>
using namespace std;

void main( )
{
  int val_e;
  cout << "Introduzca un número:";
  cin >> val_e;
  cout << "\n Ha introducido un número" << val_e << endl;
}
```

A recordar

La notación `<Intro>` se utiliza para representar la pulsación de la tecla `INTRO` (`ENTER` o `RETURN`).

Al ejecutarse el programa anterior, se podrían seguir estas acciones:

```
Introduzca un número: 4321<Intro>
Ha introducido el número 4321
```

Lectura de datos carácter

Los caracteres se leen uno a uno, de acuerdo con las reglas siguientes:

1. Las tabulaciones, nuevas líneas y avances de página son ignorados por `cin` cuando se utiliza el operador `>>`. Sin embargo, los espacios en blanco se pueden leer utilizando el operador `cin`.
2. Los valores numéricos se pueden leer como caracteres, pero cada dígito es un carácter independiente.

```
// Este programa lee datos carácter
#include <iostream> using namespace std;
void main( )
{
  char Letra1;
  char Letra2;
  char Letra3;
  cin >> Letra1 >> Letra2 >> Letra3;
  cout << Letra1 << Letra2 << Letra3;
}
```

Algunas ejecuciones del programa anterior son:

```
Letras    ABC    7547 5     47.543
Let       ABC    754        47.
```

Obsérvese que la lectura de un carácter cada vez impone una restricción en la introducción de datos carácter. Se requiere una variable independiente para cada carácter individual introducido. Por esta causa, se necesita disponer de un medio para leer datos de cadena.

Lectura de datos cadena

Cuando se utiliza el operador de extracción para lectura de datos tipo cadena, se producirán anomalías si las cadenas constan de más de una palabra separada por blancos.

Ejemplo 33.7

```
// Listado LEERCAD1.CPP
#include <iostream>
using namespace std;

int main( )
{
  float salario;
  cout << "\n Introduzca nombre, edad y salario: \n";
  cin >> nombre >> edad >> salario;
  cout << "\n Nombre:  " << nombre << "\n Edad: "
  << edad << "\n Salario: " << salario << endl;
  return 0;
}
```

Si la cadena de entrada contiene más de una palabra, el objeto `cin` no leerá más que la primera palabra, truncando el resto de la cadena.

```
#include <iostream>
using namespace std;
void main( )
{
  char Nombre[30];
  cin >> Nombre;
  cout << '\n' << Nombre;
}
```

Cuando el usuario teclea

```
Pepe Mackoy
```

el sistema visualiza

```
Pepe
```

La razón de los caracteres truncados (`Mackoy`) es que cuando se leen datos cadena, el operador `>>` hace que el objeto `cin` termine la operación de lectura, siempre que se encuentre cualquier espacio en blanco, de modo que la variable `Nombre` contiene sólo `Pepe`.

El sistema para resolver esta anomalía puede ser definir una cadena para cada palabra completa a introducir. Sin embargo, el método más eficiente consistirá en utilizar funciones miembro **`get ( )`** y **`getline ( )`**.

Funciones miembro `get ( )` y `getline ( )`

El objeto flujo `cin` contiene varias funciones miembro que permiten procesar entrada de cadenas y caracteres sin utilizar el operador de extracción. La función miembro **`get ( )`** lee un único carácter o una línea de datos del teclado (cadenas).

La función miembro `get ( )` tiene varios formatos y está definida en la clase `istream`. Desde un punto de vista práctico, `get ( )` se puede utilizar de dos modos: `get ( )`, se utiliza sin parámetros en cuyo caso el valor devuelto se puede utilizar con una referencia a un carácter; otro formato es `get` con parámetros (una referencia a un carácter).

El primer formato de `get ( )` es sin parámetros. Este formato devuelve el valor del carácter encontrado y devolverá `EOF` (fin de archivo, *end of file*) si se alcanza el final del archivo. Su prototipo de función es:

```
int get( )
```

Esta versión suele utilizarse normalmente en bucles de entrada.

Ejemplo 33.8

Lectura de caracteres con la función `cin.get ( )`.

```
#include <iostream>
using namespace std;

int main( )
{
  char c;
  while ((c = cin.get( ))!= EOF)
  {
    cout << "c:" << c << endl;
  }
  cout << "\n Terminado!\n";
  return 0;
}
```

Nota

Para salir de este programa, debe enviar un final de archivo desde el teclado. En computadoras DOS/Windows utilice `Ctrl+Z`; en sistemas UNIX utilice `Ctrl+D`.

Al ejecutar el programa se produce esta salida.

```
Hola
c : H
c : o
c : l
c : a
Mundo
c : M
c : u
c : n
c : d
c : o
(Ctrl+Z)        // o bien ^D, Ctrl+D
Terminado!
```

Explicación

Cada llamada de la función `cin.get ( )` lee un carácter más de `cin` y lo devuelve a la variable `c`. A continuación, la sentencia del interior del bucle inserta `c` en el flujo de salida. Estos caracteres se acumulan en un búfer hasta que se inserta el carácter *fin de línea.* Entonces se limpia el búfer. Cuando se encuentra `EOF (Ctrl+Z o Ctrl+D)`, se sale del bucle. En la mayoría de las computadoras `EOF` toma el valor –1.

Ejemplo 33.9

La constante entera `EOF`

```
void main( )
{
  cout << "EOF =" << endl;
}
```

Al ejecutarse

```
EOF = -1
```

Otro formato de la función `get ( )` lee el siguiente carácter del flujo de entrada en su parámetro que se pasa por referencia

```
istream& get(char& c);
```

Esta versión devuelve *nul* cuando se detecta el final del archivo, de modo que se puede utilizar para controlar un bucle de entrada tal como éste:

```
while (cin.get(car))
```

Ejemplo 33.10

Lectura de caracteres con la función `cmn.get ( )`.

```
main( )
{
    char c;
  while (cin.get(c))
    cout << c;
  cout << endl;
}
```

```
En un lugar de Sierra Magina
En un lugar de Sierra Magina
Conocido por Carchelejo
Conocido por Carchelejo
^D
```

Ejemplo 33.11

Otra aplicación de `get ( )` con parámetro referencia.

```
// Listado get() << c << endl;
#include <iostream>
using namespace std;

int main( )
  {
  char a, b, c;

    cout << "Introduzca tres letras:";
    cin.get(a).get(b).get(c);

   cout << "a:" << a << "\nb:" << b << "\nc:"
   return 0;
  }
```

Ejecución

```
Introduzca tres letras: ono
a: o
b: n
c: o
```

Explicación

Se crean tres variables de carácter. Se realizan tres llamadas a `cin.get( ) ...` y se pone la primera letra en `a` y vuelve a `cin`, de modo que se llama a `cin.get(b)` y se pone la siguiente letra en `b`. El resultado final es que se llama a `cin.get(c)` y se pone la tercera letra en `c`.

Existe un tercer formato de la función `get ( )`, similar a la función `getline ( )` que se verá posteriormente. Su prototipo es

```
istream& get (char* bufer, int n, char sep = '\n');
```

Este formato lee caracteres del búfer hasta que se lean, o bien $n - 1$ caracteres, o bien hasta que se encuentre el carácter separador *sep*.

Regla del prototipo de `get ( )`
El primer parámetro es un puntero a un arreglo de caracteres; el segundo parámetro es el número máximo de caracteres a leer más uno y el tercer parámetro es el carácter de terminación.

Ejemplos

```
char cadena[80];                    // Array de entrada
cin.get(cadena, 80);                // Lectura de caracteres hasta que
                                    // se encuentra un carácter nueva
                                    // línea o se
                                    // han leído 79 caracteres
```

Ejemplo 33.12

Lectura de caracteres con `cin.get ( )`

```
main( )
{
  char bufer[80];
  cin.get(bufer, 8);                 // lee 7 caracteres del bufer
  cout << "[" << búfer <<     "]\n";
  cin.get(bufer, sizeof(bufer));
  cout << "[" << búfer <<     "]\n";
}
```

Si el búfer almacena `ABCDE|FGHIJ|KLMNO|PQRST|UVWXY|Z`

la salida será:

```
[ABCDE|F]
[GHIJ|KLMNO|PQRST|UVWXY|Z]
```

La función `getline`

Otra función miembro que se puede utilizar para la lectura de datos es `getline ( )`. La función `getline ( )` permite a `cin` leer cadenas completas, incluyendo espacios en blanco; es muy similar a la función miembro `get ( )` de dos o tres argumentos, excepto que el carácter de terminación en `getline ( )` se extrae del flujo de entrada y se considera. El formato de `getline ( )` es

```
cin.getline(var_cad, long_max_cad+2, car_separador);
```

La función `getline ( )` utiliza tres argumentos. El primer argumento, *var_cad*, es el identificador de la variable cadena. El segundo argumento es la máxima longitud de la cadena (número máximo de caracteres que se leen); la longitud ha de ser mayor que la cadena real al menos en dos

caracteres, para permitir los caracteres `'\n' (CRLF)` y `'\ 0'` (carácter nulo). El carácter de separación se lee y almacena como el siguiente al último carácter de la cadena. La función `getline ( )` inserta automáticamente el carácter de terminación *nulo* como el último carácter de la cadena. Si no se especifica ningún carácter de terminación, se toma de modo determinado el carácter `'\ n'`.

Veamos un programa ejemplo:

```
// Uso de cin y getline para leer datos de cadena

#include <iostream>
using namespace std;

void main( )
{
  char Nombre[40];
  cin.getline(Nombre, 40);
  cout << Nombre;
}
```

Al ejecutar el programa con la entrada `Sierra de Cazorla` la variable `Nombre` acepta toda la cadena, es decir, todos los caracteres incluyendo blanco.

Ejemplo 33.13

```
// Este programa lee y escribe
// Nombre, Dirección y Teléfono del usuario

#include <iostream>
using namespace std;

void main( )
{
  //Definición de arrays de caracteres
  char Nombre[40];
  char Calle[30];
  char Ciudad[30];
  char Provincia[30];
  char CodigoPostal[5];
  char Telefono[10];

  //Lectura de datos
  cin.getline(Nombre, 40);
  cin.getline(Calle, 30);
  cin.getline(Ciudad, 30);
  cin.getline(Provincia, 30);
  cin.getline(CodigoPostal, 5);
  cin.getline(Telefono, 10);

  //Visualizar datos
  cout << Nombre;
  cout << Calle;
  cout << Ciudad;
  cout << Provincia;
  cout << CodigoPostal;
  cout << Telefono;
}
```

Una entrada de datos en una ejecución del programa:

```
Luis Enrique
Santiago Bernabéu 45
Madrid
Madrid
28230
91-7151515
```

A recordar

La diferencia entre `get ( )` y `getline ( )` es que esta última almacena el carácter separador o delimitador en la cadena antes de añadir el carácter nulo.

producirá una salida tal como:

```
Luis Enrique
Santiago Bernabéu 45
Madrid
Madrid
28230
91-7151515
```

Problemas en la utilización de `getline( )`

Aunque `getline ( )` funciona cuando se leen datos tipo de cadena de modo consecutivo, se presentarán problemas cuando se intenta utilizar una variable de cadena, después de que se ha utilizado `cin` para leer una variable carácter o numérica. Por ejemplo, supongamos un programa como el siguiente:

```
// PERGETL.CPP
// Programa que muestra el problema de utilizar cin.getline( )
// para leer una cadena después de haber leído una
// variable numérica o carácter

#include <iostream>
using namespace std;

void main( )
{
  char Nombre[30];
  int Edad;

  cout << "Introduzca edad:";
  cin << Edad;
  cout << Edad;

  cout << "Introduzca el nombre:";
  cin.getline(Nombre, 30);
  cout << Nombre;
}
```

Si se introducen en `Edad` y `Nombre` los valores `525` y `Mortimer`; es decir, suprimiendo la salida del programa siguiente:

```
Introduzca edad: 525
Introduzca el nombre: Mortimer
```

Los valores que toman las variables citadas son:

```
Edad 525
Nombre '\n''\0'
```

La razón de los valores anteriores se debe a que al introducir `525`, se debe pulsar la tecla `<INTRO>(ENTER)` a la terminación. Esta acción inserta un carácter `CRLF` (retorno de carro/avance de línea) y permanece en el búfer (memoria intermedia). Cuando la sentencia `cin.getline (Nombre, 30)` se ejecuta, se lee la memoria intermedia del teclado y se encuentra el carácter `CRLF`, dado que éste es, en forma determinada, el carácter de separación, se detiene la lectura y se inserta el carácter de terminación nulo en el arreglo. Por consiguiente, no se puede introducir el nombre.

Existen tres métodos para resolver el problema:

1. Especificar un carácter de separación diferente en la función `getline ( )`. El usuario debe introducir este carácter y se almacenará como último carácter, antes del carácter nulo del arreglo.
2. Limpiar la memoria intermedia (búfer) del teclado leyendo el carácter sobrante `CRLF` en una variable *basura*, después de leer cualquier dato numérico o carácter y antes de leer cualquier dato cadena. De este modo en la variable basura (auxiliar) se almacenarían los datos de la memoria intermedia y ya se podrían leer los caracteres útiles para la variable cadena.

```
// LIMPIARB.CPP
// Limpieza de la memoria intermedia con una
// variable auxiliar o basura

#include <iostream.h>
using namespace std;

void main( )
{
  char Auxiliar[2];
  char Nombre[30];
  int Edad;

  cout << "Introduzca edad:";
  cin >> Edad;                              // Lectura de datos numéricos
  cout << Edad;
  cin.getline(Auxiliar, 2);                 // Limpiar búfer de teclado
  cout << "Introduzca el nombre:";
  cin.getline(Nombre, 30);                  // Leer datos de cadena
  cout << Nombre;
}
```

3. Utilizar una sentencia de lectura diferente; para ello se recurre a funciones de cadena de `E/S` definidas en la biblioteca estándar de C y C++: `gets( )` y `fgets( )`. Estas funciones se encuentran dentro del archivo de cabecera `stdio.h`.

Formateado[1] de salida

Si no se instruye al operador de inserción para realizar operaciones de formateado específico, la salida se formatea cuando se convierte un dato a un flujo de caracteres para la salida. La tabla 33.3 describe cómo formatea la salida el operador de inserción para diversos tipos de datos.

Ejemplos

```
cout << "CAZORLA#";
char letra = 'J';
cout << letra;

float f = 123.456789101, g = 1234567890.456;
cout << f << '#\n';
cout << g << '#\n';
int e = 123, en = -525;
cout << e << en << '\n';
```

Si se ejecutan las sentencias `cout` precedentes se produce una salida tal como ésta:

```
CAZORLA#J123.456789#
1.234567E+009#
123-5 25
```

Manipuladores

La entrada y salida de datos realizada mediante los operadores de inserción y extracción puede formatearse ajustándola a izquierda o derecha, proporcionando una longitud mínima o máxima, precisión, etc.

La solución al problema son los *manipuladores* que son funciones especiales diseñadas específicamente para modificar el funcionamiento de un flujo. Los *manipuladores* manipulan el formato de un objeto. La biblioteca `iostream` viene con un número de manipuladores incorporados, aunque pueden añadirse otros fácilmente. La mayoría de las veces, los manipuladores se utilizan para indicar for-

[1] La 22a. edición (2001) del *Diccionario de la Lengua Española* de la Real Academia Española incorpora el verbo **formatear** como acepción informática.

Tabla 33.3 Conversión para salida de tipos de datos.

Tipo	Tipo de conversión de salida
`char`	Los caracteres imprimibles se visualizan con una anchura de una columna. Los caracteres de control, tales como nueva línea, tabulación, etc., pueden producir más caracteres de salida.
`int`	Cualquier tipo entero (`int`, `short` o `long`) se visualizan como números decimales con anchura suficiente para contener el número y un signo menos si el entero es negativo.
`cadena`	La anchura en pantalla es igual a la longitud de la cadena.
`float`	Los números reales en coma flotante se visualizan con seis dígitos decimales de precisión. Los ceros no significativos no se visualizan. Si el número es muy grande o muy pequeño, se visualiza el número con un exponente prefijado por la letra E y dos dígitos (o tres dígitos si el tipo es `double`). La anchura siempre es lo suficiente para mantener un signo menos y/o un exponente.

mateado, tal como la anchura de un campo, la precisión en números de coma flotante, etc. Sin embargo, los manipuladores no sólo pueden realizar formateado, sino otras tareas. Normalmente los manipuladores se utilizan en el centro de una secuencia de inserción o extracción de flujos. La mayoría de los manipuladores no tienen argumentos y están diseñados para realizar la tarea de formateado lo más sencilla posible.

Los manipuladores de flujos están incluidos, fundamentalmente, en el archivo de cabecera `ioma - nip`. Los manipuladores se pueden utilizar tanto para flujos de entrada como de salida. Consulte el manual de referencia de su compilador C++ para cualquier ampliación de la información de este capítulo.

La tabla 33.4 recoge los manipuladores de flujo de E/S. Cualquiera de los manipuladores se puede insertar en la sentencia `cout` como cualquier otro elemento.

Tabla 33.4 Manipuladores de flujos.

Manipulador	Acción
`dec`	Utiliza conversión decimal (*de modo predeterminado*).
`hex`	Utiliza conversión hexadecimal.
`oct`	Utiliza conversión octal.
`ws`	Extrae caracteres espacios en blanco.
`endl`	Inserta nueva línea (se puede utilizar en lugar de `'\n'`).
`ends`	Añade un carácter terminal nulo al flujo de salida (`'\0'`).
`flush`	Limpia (fluye) un flujo de salida.
`setbase (n)`	Establece la base de conversión a *n* (0, 8, 10 o bien 16). 0 significa decimal por defecto.
`setprecision (n)`	Establece la precisión de coma flotante a *n*.
`setw (n)`	Establece la anchura del campo a *n*.
`setf ill (c)`	Establece el carácter de relleno a *c*.
`setiosflags (f)`	Establece los bits de formato especificado por el argumento *f* de tipo `long`.
`resetiosflags (f)`	Pone a cero los bits de formato especificados por el argumento *f* de tipo `long`.

La sintaxis típica para utilizar los manipuladores es:

```
cout << setw (anchura del campo) << elemento de salida;
```

requiriendo la inclusión del archivo de cabecera *iomanip*

```
#include <iostream>
#include <iosmanip> ...
cout << setw(3) << i << setw(5) << i*i*i;
```

Bases de numeración

Normalmente, los enteros se visualizan como decimales (números escritos en base 10); es posible, sin embargo, seleccionar una base de numeración distinta (octal, 8, hexadecimal, 16) llamando a las funciones (manipuladores) `dec`, `hex` o bien `oct`. Así, por ejemplo,

```
cout << dec << Total << endl;
```

visualiza el valor de `Total` en base 10. `dec` es importante para volver a seleccionar la base 10 (decimal) después de haber trabajado con otras bases. Por ejemplo, si `Total` toma el valor decimal 255, la sentencia

```
cout << hex << Total << endl;
```

produce la salida

```
ff valor hexadecimal correspondiente a 255 decimal
```

Se pueden entremezclar en una misma sentencia diversos manipuladores:

```
cout << "Base 10 =" << dec << Total
     << "Base 16 =" << hex << Total << endl;
```

Ejemplo 33.14

```
// Uso de los manipuladores
#include <iostream>
using namespace std;

#include <iomanip>
using namespace std;

void main( )
{
  cout << "Valor hex de" << 40 << "en decimal es:"
       << hex << 40 << endl;
  cout << "Valor octal de" << hex << 34 << "en hexadecimal es:"
       << oct << 34 << endl;
  cout << dec; // Se restablece la numeración decimal
}
```

La salida del programa es:

```
Valor hex de 40 en decimal es: 28
Valor octal de 22 en hexadecimal es: 42
```

ya que `40` decimal equivale a `28 (2 * 16 + 8 = 40)` en hexadecimal, y `42` en base octal `(4 * 8 + 2 = 34)` equivale a `22` en hexadecimal `(2 * 16 + 2 = 34)`.

El manipulador `setbase (int n)` establece la base numérica a 8, 10 o 16. Este manipulador parametrizado funciona igual que los manipuladores 8, 10 o 16. Un ejemplo puede ser

```
cout << setbase(10);
cin >> setbase(10);
```

Ejemplo 33.15

```
#include <iostream>
#include <iomanip>
using namespace std;
void main( )
{
  const int n = 100;

  // visualizar los resultados en distintas bases
  cout << "\n" << n << " "
       << oct << n << " "
       << hex << n << endl;
 }
```

Anchura de los campos

El manipulador `setw ( )` proporciona un medio para fijar la anchura del formato de salida. En forma predeterminada, los datos que entran y salen se corresponden con el espacio necesario para almacenar ese dato, es decir, si se escribe una cadena de seis caracteres, la anchura de salida es 6. El prototipo de `setw` es

```
setw(int n)
```

Un ejemplo del uso de `setw( )` para visualizar un campo de ocho caracteres de ancho es

```
cout << "12345678901234567890" << endl;
cout << setw(8) << "Hola";
cout << "Mundo";
```

que produce la salida siguiente:

```
12345678901234567890
   Hola mundo
```

Es decir, el ancho del campo `Hola` ha sido ocho caracteres y se alinea a derechas.

Ejemplo 33.16

```
// ANCHURA.CPP

#include <iostream>
using namespace std;

#include <iomanip>

void main( )
{
  cout << setw(10) << "M" <<setw(10) <<    "N" << endl;
  cout << setw(10) <<   1 <<setw(10) <<   7.77 << endl;
  cout << setw(10) <<  10 <<setw(10) <<  77.77 << endl;
  cout << setw(10) << 100 <<setw(10) << 777.77 << endl;
}
```

La salida será

```
         M         N
         1      7.77
        10     77.77
       100    777.77
```

Se puede utilizar la función miembro `width( )` para modificar la anchura del campo. El valor del parámetro pasado será la anchura en caracteres. Si se especifica una anchura en la entrada, lo que se hace es limitar la entrada a esa longitud. Así,

```
cout.width(5);
cout << "ABC" << "DEF" << "GHI"
```

visualiza

```
ABC DEF GHI
```

y

```
cin.width(20);
```

limita la entrada a 20 caracteres.

Rellenado de caracteres

Siempre que se establece la anchura de un campo más grande que la anchura de los datos, los espacios adicionales se rellenan con caracteres blancos.

Se puede cambiar el carácter de relleno utilizando la función miembro `fill( )`. Así, por ejemplo, supongamos que se desea que el carácter de relleno sea un asterisco (`*`); un código fuente que realiza esa tarea puede ser:

```
cout << "12345678901234567890" << endl;
cout.width(15);
cout.fill('*');
cout << "Hola Mackoy" << endl;
float Z = 99.99;
cout.width(15);
cout << Z;
```

cuya salida será:

```
12345678901234567890
****Hola Mackoy
******99.989998
```

Obsérvese que el carácter de relleno permanece hasta que se vuelve a cambiar.

```
char Relleno;

Relleno = cout.fill('*');
cout << "Hola Mackoy" << endl;
cout.fill(Relleno); // Se recupera el antiguo carácter de relleno
```

Precisión de números reales

Si se visualiza un número en coma flotante, se visualizan hasta seis dígitos de precisión, en forma predeterminada. Los ceros a la derecha del punto decimal se suprimen. Se puede cambiar el número de dígitos de precisión de seis a otro valor con el manipulador `setprecision( )`. Su argumento entero (*n*) especifica el número de dígitos significativos que se visualizarán. El formato es:

```
cout << setprecision(i);
```

Ejemplo 33.17

```
// Archivo PRECISIO.CPP
// Fija el número de posiciones decimales
```

```
#include <iostream>
using namespace std;

#include <iomanip>

void main( )
{
 float prueba = 814.159265;
   cout  << setprecision(2)   //2 dígitos significativos
         << prueba << endl;
   cout  << setprecision(3)   //3 dígitos significativos
         << prueba << endl;
   cout  << setprecision(4)   //4 dígitos significativos
         << prueba << endl;
   cout  << setprecision(5)   //5 dígitos significativos
         << prueba << endl;
}
```

La salida del programa es:

```
814.16
814.159
814.1592
814.15924
```

Se puede utilizar también la función miembro `precision ( )` para establecer la precisión. Por ejemplo, la sentencia fija la precisión a 3, para la salida correspondiente:

```
cout.precision(3);
```

Indicadores de formato

Cada flujo, es decir, cada objeto de la clase `istream` y `ostream` contiene un conjunto de informaciones (*indicadores*) que especifican cúal es en un momento dado su "estatuto de formato". Este modo de proceder es muy diferente del empleado por las funciones de C, tales como **`printf`** o **`scanf`**, en las que a cada operación de entrada/salida se le proporcionan los indicadores de formateado apropiado.

El método empleado por C++ es más eficiente que el empleado por C, ya que permite al usuario, eventualmente, ignorar por completo el método empleado por C, un tanto complejo por otra parte.

Cada uno de estos indicadores (*flags*) pueden establecerse o reinicializarse utilizando un manipulador incorporado. Los manipuladores `setiosflags (long)` y `resetiosflags (long)` realizan fundamentalmente estas tareas.

Uso de `setiosflags( )` y `resetiosflags( )`

El manipulador `setiosflags ( )` se define como

```
setiosflags (long f)
```

y sirve para especificar, por ejemplo, si un dato se alinea a izquierda o derecha en un campo. En forma predeterminada, los valores se alinean a derecha en un campo. La siguiente sentencia activa la opción de alineación a izquierda:

```
cout << setiosflags(ios::left);
```

Los argumentos de los manipuladores (bits indicadores de `ios`) realizan diversas tareas que se recogen en la tabla 33.5.

Se debe hacer preceder a cada argumento o *bit de estado* de la cláusula `ios::`, que significará su asociación con la clase `ios`. Por ejemplo,

```
float pi = 3.14159;
cout << setiosflags(ios::fixed) << pi << endl;
```

Tabla 33.5 Bits de estado (palabra de estado del formateado).

Bit de estado (argumento)	Propósito
`skipws`	Salta espacios en blanco en operaciones de entrada.
`left`	Justifica la salida a la izquierda del campo.
`right`	Justifica la salida a la derecha del campo.
`internal`	Rellena el campo después del signo o el indicador base.
`dec`	Activa conversión decimal.
`oct`	Activa conversión octal.
`hex`	Activa conversión hexadecimal.

`showbase`	Visualiza el indicador de base numérica.
	Ejemplo:
	`044 (número octal)`
	`0x2ea7 (número hex)`

`showpoint`	Visualiza punto decimal en valores de coma flotante. *Ejemplo:* `456.00`
`uppercase`	Visualiza valores hexadecimales en mayúsculas. *Ejemplo:* `4BFE`
`showpos`	Visualiza números enteros positivos precedidos del signo +.
`scientific`	Notación científica en los números en coma flotante. *Ejemplo:* `3.1416e+00`
`fixed`	Utiliza notación en coma fija para números en coma flotante. *Ejemplo:* `1234.5`
`unitbuf`	Vacía (limpia) las memorias (*búfers*) después de cada escritura.
`stdio`	Vacía (limpia) las memorias intermedias después de cada escritura sobre `stdout` o `stderr`.

ha seleccionado un valor de coma flotante con notación fija y la salida será:

```
3.14159
```

Se selecciona la notación científica utilizando la sentencia siguiente:

```
cout << setiosflags(ios::scientific) << pi << endl;
```

y se visualiza la salida siguiente:

```
3.14159e+00
```

Se pueden establecer múltiples indicadores en una operación mediante operadores `OR`. Por ejemplo,

```
cout << setiosflags(ios::dec|ios::showbase) << Total << endl;
```

Para limpiar o reponer los indicadores de estado, se deben utilizar `resetiosflags( )`. Por ejemplo, para limpiar o borrar el parámetro `showbase`, escribir

```
cout << resetiosflags(ios::showbase) << Total << endl;
```

Así por ejemplo, si se activa la opción de alineación a la izquierda

```
cout << setiosflags(ios::left);
```

se desactivará con la sentencia

```
cout << resetiosflags(ios::left);
```

Otro ejemplo para visualizar resultados en diferentes bases de numeración son las siguientes líneas de código:

```
cout << setiosflags(ios::showbase)
     << "\n" << v << " "
     << oct  << v << " "
     << hex  << v << endl;
```

que produce la salida:

```
100 0144    0x64
```

Ejemplo 33.18

El programa `FORMATO1.CPP` muestra un sistema para formatear datos de salida de diversas maneras.

```
//Archivo FORMATO1.CPP

#include <iostream>
using namespace std;
#include <iomanip>
void main( )
{
  float v1 = 4500.25;
  float v2 = 325.99;
  float v3 = 54225.00;

  cout  << setiosflags(ios::showpoint|ios::fixed)
        << setprecision( 2)
        << setfill('*')
        << setiosflags(ios::right);
  cout  << "\n Saldo Final: $" << setw(10) << v1 << endl;
  cout  << "\n Saldo Final: $" << setw(10) << v2 << endl;
  cout  << "\n Saldo Final: $" << setw(10) << v3 << endl;
}
```

Al ejecutar el programa se produce:

```
Saldo Final: $***4500.25
Saldo Final: $****325.99
Saldo Final: $**54225.00
```

Ejemplo 33.19

```
// Archivo FORMATOS2.CPP

#include <iostream>
using namespace std;

#include <iomanip>

void main( )
{
  const float p = 3.14159;
  // Visualizar números reales con diversos manipuladores
  cout << setiosflags(ios::showpos|ios::scientific)
       << "\n El valor de PI es"
```

```
        << setprecision( 3)
        << setw(15) << setfill('*')
        << setiosflags(ios::right)
        << p;
}
```

La salida de este programa es

```
El valor de PI es*****+3.142e+00
```

Las funciones miembro `setf ( )` y `unsetf ( )`

Existe un segundo método para establecer los indicadores de flujo: llamar a las funciones miembro **`setf ( )`** y **`unsetf ( )`**. Estas funciones son similares a los manipuladores `setiosflags( )` y `resetiosflags( )`. La diferencia reside en que **`setf ( )`** y **`unsetf ( )`** son verdaderas funciones miembro. Se accede a las funciones miembro directamente:

```
cout.setf(ios:: scientific);
cout.unsetf(ios::scientific);
```

Archivos C++

C++ utiliza flujos (*streams*) para gestionar flujos de datos, incluyendo el flujo de entrada y de salida. Un **archivo** es una secuencia de bits almacenados en algún dispositivo externo tal como un disco o una cinta magnética. Los bits se interpretan de acuerdo con el protocolo de algún sistema software. Si estos bits se agrupan en bytes de 8 bits interpretados por el código ASCII, entonces el archivo se denomina *archivo de texto* y puede ser procesado por editores estándar. Si los bits se agrupan en palabras de 32 bits representando a pixeles de colores, entonces el archivo es un *archivo gráfico* y se procesa por un software de gráfico especializado. Si el archivo es un programa ejecutable, entonces sus bits se interpretan como instrucciones al procesador de la computadora.

En C++, un archivo es simplemente un flujo externo: una secuencia de bytes almacenados en disco. Si el archivo se abre para salida, es un flujo de archivo de salida. Si el archivo se abre para entrada, es un flujo de archivo de entrada.

La biblioteca de flujos contiene tres clases, `ifstream`, `ofstream` y `fstream`, y métodos asociados para crear archivos y manejo de entrada y salida de archivos.

Las tres clases, `ifstream`, `ofstream` y `fstream`, se declaran en el archivo de cabecera *fstream,* que, incidentalmente, incluye el archivo de cabecera *iostream* de modo que no necesita definir explícitamente `#include <iostream` si se incluye el archivo de cabecera *fstream* `(#include <fstream>)`.

C++ admite dos tipos de archivos: texto y binario. Los *archivos de texto* almacenan datos como códigos ASCII. Los valores simples, tales como números y caracteres únicos, están separados por espacios. Los archivos de texto se pueden utilizar para almacenamiento de datos o crear imágenes de salida impresa que se pueden enviar más tarde a una impresora.

Los *archivos binarios* almacenan flujos de bits, sin prestar atención a los códigos ASCII o a la separación de espacios. Son adecuados para almacenar objetos. Sin embargo, el uso de los archivos binarios requiere usar la dirección de una posición de almacenamiento.

Apertura de archivos

Antes de que un programa pueda leer o escribir de un disco, se debe *abrir* el archivo. El proceso de la *apertura de un archivo* identifica la posición del archivo en el programa. Para abrir un archivo de texto C++ para lectura, se crea un objeto (un flujo) de la clase `ifstream`; para abrir un archivo para

escritura, se crea un objeto de la clase `of stream`. Se puede entonces utilizar los nombres de los flujos que se crean con los operadores de inserción y extracción.

Al igual que los flujos, en forma predeterminada, `cin` y `cout`, los flujos de E/S de archivos ANSI pueden transferir datos sólo en una dirección. Esto significa que se deben abrir y manipular flujos independientes para lectura y escritura de archivos de texto. La biblioteca de flujos C++ ofrece un conjunto de funciones miembro comunes a todas las operaciones de E/S de flujo de archivo.

Para *abrir el archivo para lectura* al comienzo de cada ejecución de programa, el programa incluye la sentencia

```
ifstream aen("demo");
```

El objeto `ifstream` se llama `aen`. Está asociado con un archivo cuyo nombre del camino está dentro de la lista de parámetros. Este parámetro se pasa al constructor de clase, que se utiliza para localizar y abrir el archivo.

La *apertura de un archivo para escritura o salida* se realiza con la sentencia `ofstream` definiendo un objeto de la clase `ofstream` (*output file stream*).

```
ofstream archsal ("copy.out", ios_base::out);
```

Un archivo `ofstream` se puede abrir de dos maneras: 1) en modo salida `(ios_base::out)`; 2) en modo añadir `(ios_base::app)`. En forma predeterminada, un archivo `ofstream` se abre en modo salida. La definición de `archsa1` es equivalente a la del archivo de salida.

```
// abierta en modo salida por defecto
ofstream archsalida("copy.sal");
```

Si un archivo existe se abre en modo salida, todos los datos almacenados en ese archivo se descartan. Si se desea *añadir* en lugar de *reemplazar* los datos dentro de un archivo existente, se debe abrir el archivo en modo *append* ("añadir"). Los datos adicionales escritos en el archivo se añaden a continuación de su extremo final. *De cualquier manera, si el archivo no existe, se creará.*

Precaución

Antes de intentar leer o escribir en un archivo, es siempre una buena idea verificar que se ha abierto con éxito. Se puede comprobar `archsal` del modo siguiente:

```
if (!archsal) { // apertura fallida
  cerr << "no se puede abrir copy.sal para salida\n";
  exit (-1);
}
```

Ejercicio 33.1

El siguiente programa obtiene caracteres de la entrada estándar y los pone en el archivo `copy.sal`.

```
#include <fstream>

int main( )
{
  // abrir un archivo copi.sal para salida;
  of stream archsal ("copi.sal");

  if (!archsal) {
    cerr << "No se puede abrir copi.sal para salida:" << endl;
      return -1;
  }
  char car;
  while (cin.get (car))
    archsal.put (car);
  return 0;
}
```

Apertura de un archivo sólo para entrada

Leer un archivo especificado por el usuario y escribir su contenido en la salida.

Para esta operación se utiliza un objeto de `ifstream`. La clase `ifstream` se deriva de `istream`.

Ejercicio 33.2

```
cerr << "incapaz abrir archivo de entrada:"
  <<nombre_arch << "precaución de salida\n";
```

```
#include <fstream>
using namespace std;
```

```
int main( )
{
  char nombre_arch[80];
  cout << "Nombre archivo: "; cin >> nombre_arch;
  ifstream ArchEn(nombre_arch);
  if (!ArchEn) {
    cerr << "No se puede abrir: " << nombre_arch;
    return -1;
  }
  char car;
  while (ArchEn.get(car))
    cout.put (car);
  return 0;
}
```

Un archivo se puede desconectar del programa invocando la función miembro `close ( )`. Por ejemplo,

```
ArchivoAct.close();
```

E/S en archivos

El sistema de E/S de C++ se puede utilizar para hacer E/S a archivos. Para poder realizar las operaciones de E/S, se necesita incluir el archivo de cabecera *fstream* en el que se definen varias clases con diferentes funciones miembro. El archivo de cabecera `<fstream>` declara las clases `ifstream`, `ofstream` y `fstream`.

Específicamente la clase `ifstream` se deriva de la clase `istream` y permite a los usuarios acceder a archivos y leer datos de ellos. La clase `ofstream` se deriva de la clase `ostream` y permite a los usuarios acceder a archivos y escribir datos en ellos. Por último, la clase `fstream` se deriva de las clases `ifstream` y `ofstream` y permite a los usuarios acceder a los archivos para entrada y salida de datos. Para abrir y gestionar adecuadamente las clases `ifstream` y `ofstream` relacionadas con un sistema de archivos, se debe declarar con un constructor apropiado.

```
ifstream( );
ifstream (const char*, int = ios::in, int prot = filebuf::openprot);
ofstream( );
ofstream (const char*, int = ios::out, int prot = filebuf::openprot);
```

El constructor sin argumentos crea una variable que se asociará posteriormente con un archivo de entrada. El constructor de tres argumentos toma como su primer argumento el archivo con nombre. El segundo argumento especifica el modo archivo. El tercer argumento es para protección de archivos.

- La apertura de un flujo de entrada se debe declarar en la clase `ifstream`.
- La apertura de un flujo de salida se debe declarar en la clase `ofstream`.
- Los flujos que realicen operaciones de entrada y salida deben declararse en la clase `fstream`.

Ejemplo

```
ifstream ent;       // crea un flujo de entrada
ofstream sal;       // crea un flujo de salida
fstream ambos;      // crea un flujo de entrada y salida
```

La función `open`

Una vez creado el flujo, se puede utilizar la función `open ( )` para asociarlo con un archivo. Esta función es miembro de las tres clases de flujo. La declaración de la función `open ( )` (su prototipo) tiene por prototipos:

```
void open (const char*, int = ios::in, int prot = filebuf::openprot);
            // abre archivos ifstream
void open (const char*, int = ios::out, int prot = filebuf::openprot);
            // abre archivo ofstream
```

Estas funciones se pueden utilizar para abrir y cerrar archivos apropiados. El prototipo estándar es:

```
void open(const char*nomarch, int modo, int acceso = (filebuf::openprot);
```

nombre del archivo (puede incluir un especificador de vía de acceso)

determina cómo se abrirá el archivo

protección del archivo

Los posibles valores del argumento *modo* se definen como enumeradores de la clase `ios` (tabla 33.6).

El argumento *acceso* especifica el permiso de acceso en forma predeterminada que se asigna a un archivo si se crea por la función constructor. El valor en forma predeterminada es `filebuf::openprot`.

Ejemplos

```
// Ejemplo 1. Abre el archivo AUTOEXEC.BAT para entrada de texto
char* cAutoExec = "\\AUTOEXEC.BAT";
fstream f;
// abrir para entrada
f.open (cAutoExec, ios::in);

// Ejemplo 2. Abre el archivo DEMO.DAT para salida de texto
fstream f;
// abrir para salida
f.open ("DEMO.DAT", ios::out);

// Ejemplo 3. Abre el archivo PRUEBAS.DAT para entrada y
// salida binaria
fstream f;
// abrir para E/S de acceso aleatorio
f.open ("PRUEBAS.DAT", ios::in | ios::out | ios::binary);
```

Tabla 33.6 Valores del argumento modo de archivo.

Argumento	Modo
`ios::in`	Modo entrada
`ios::app`	Modo añadir
`ios::out`	Modo salida
`ios::ate`	Abrir y buscar el fin del archivo
`ios::nocreate`	Genera un error si no existe el archivo
`ios::trunc`	Trunca el archivo a 0 si existe ya
`ios::noreplace`	Genera un error si el archivo existe ya
`ios::binary`	El archivo se abre en modo binario

Consejo

A fin de abrir un flujo para entrada y salida, se deben especificar los dos valores del argumento *modo*, `ios::in` e `ios::out`, como se muestra en el siguiente ejemplo:

```
fstream miflujo;
miflujo.open ("prueba", ios::in |ios::out);
```

Si `open ( )` falla, el flujo será nulo. A pesar de que es sintácticamente válido abrir un archivo mediante la función `open ( )`, no suele hacerse, ya que las clases `ifstream`, `of stream` y `fstream` disponen de funciones constructor que abren automáticamente el archivo. Las funciones constructor tienen los mismos parámetros y valores por omisión que la función `open( )`. En consecuencia, la forma habitual de abrir un archivo será la siguiente:

```
ifstream miflujo ("miarch"); // abre el archivo para entrada
```

Si no se puede abrir el archivo por algún motivo, el valor de la variable de flujo asociada `miflujo` será cero. Se puede utilizar el siguiente código para confirmar que el archivo se ha abierto correctamente:

```
ifstream miflujo ("archdemo");
if (!miflujo) {
  cout << "No se puede abrir el archivo\n"; // error
}
```

Ejercicio 33.3

El siguiente ejemplo muestra el uso de las clases `ifstream` y `ofstream`. En el ejemplo, un archivo denominado fuente se abre para lectura y otro archivo que se denomina destino se abre para escritura. Si ambos archivos se abren con éxito, el contenido del archivo fuente se copia en el archivo destino. Si un archivo no se puede abrir con éxito, se señala un error:

```
ifstream fuenteF( "fuente");
ofstream destinoF( "destino");
if(!fuenteF || !destinoF)
  cerr << "Error fuente o destino open failed\n";
else
  for(char c = 0; destinoF && fuenteF.get(c);)
     destinoF.put(c);
```

La función `close`

La función miembro `close` cierra el archivo al que se conecta el objeto de flujo. La sintaxis de `close` es:

```
void close( );
```

Ejemplo

```
char cAutoExec = "AUTOEXEC.BAT";
fstream f;
// abrir para entrada
f.open(cAutoExec, ios::in);
// sentencias de E/S
f.close()       // cerrar bufer del flujo del archivo
```

A recordar

La función `close ( )` no utiliza parámetros ni devuelve ningún valor.

Este ejemplo abre el archivo `AUTOEXEC.BAT` para entrada, realiza operaciones de entrada y, a continuación, cierra el búfer del flujo del archivo.

Otras funciones miembro

Además de las funciones miembro `open` y `close`, la biblioteca de flujos C++ ofrece las siguientes funciones miembro y operadores:

- La función miembro `good`, que devuelve un valor distinto de cero si no existe ningún error en una operación de flujo. La declaración de esta función miembro es:

```
int good( );
```

- La función miembro `fail`, que devuelve un valor distinto de cero si existe un error en una operación de flujo. La declaración de esta función miembro es:

```
int fail( );
```

- La función miembro `eof`, que devuelve un valor distinto de cero si el flujo ha alcanzado el final del archivo. La declaración de esta función miembro es:

```
int eof( );
```

- El operador sobrecargado `!`, que determina el estado del error. Este operador toma un objeto de flujo como un argumento.

Además de las funciones miembro heredadas de las clases `iostream`. Las clases `ifstream`, `ofstream` y `fstream` definen también sus propias funciones específicas:

`void attach (int fd)`	Conecta el objeto flujo al flujo referenciado por el descriptor del archivo `fd`.
`filebuf* rdbuf ( )`	Devuelve un arreglo `filebuf` asociado con el objeto flujo.

Lectura y escritura de archivos de texto

La lectura y escritura en un archivo de texto se puede realizar con los operadores `<<` y `>>` con el flujo abierto.

Ejercicio 33.4

El siguiente programa escribe un entero y un valor en coma flotante y una cadena en un archivo llamado `DEMO`.

```
#include <iostream>
using namespace std;

#include <fstream>
int main( )
{
  ofstream sal ("demo");
  if (!sal) {
    cerr << "No se puede abrir el archivo" << endl;
    return -1;
  }

  sal << 10 << " " << 325.45 << endl;
  sal << "Ejemplo de archivo de texto" << endl;

  sal. close ( );
  return 0;

}
```

Ejercicio 33.5

El programa siguiente lee un número entero, un número en coma flotante, un carácter y una cadena del archivo `DEMO`.

```
#include <iostream>
#include <fstream>

int main( )
{
  char c;
  int i;
  float f;
  char cad[40];

  ifstream ent ("demo");
  if (!ent)
  {
    cerr << "No se puede abrir el archivo" << endl;
    return -1;
  }
  ent >> i;
  ent >> f;
  ent >> c;
  ent >> cad;

  ent.close( );
  return 0;
}
```

A recordar

El operador >> produce en la lectura de archivos de texto una conversión de ciertos caracteres. Por ejemplo, se omiten los caracteres en blanco.

Ejercicio 33.6

Escribir datos en un archivo de datos externo.

El archivo de cabecera `<fstream>` define la clase `of stream` que debe ser *instanciada* para escribir en un archivo externo.

Es preciso abrir un archivo `notas.dat` como archivo de salida, construir el flujo de salida `archsal` y conectar ese flujo al archivo. Estas operaciones se realizan invocando al constructor de la clase `ofstream`.

El algoritmo de escritura en un archivo de datos externo prosigue asegurando que el archivo se ha abierto adecuadamente. Para ello se invoca al operador de negación sobrecargado `(!archsal)` y en caso de que se produzca un error, se visualiza un mensaje de error y se termina el programa.

```
cerr << "Error: no se puede abrir archivo de Salida" << endl;
exit(1)
```

Si el proceso es correcto, el programa utiliza un bucle de entrada para leer nombres, identificar los números de expediente del alumno (`exp`) y las calificaciones (`notas`) de la entrada estándar y escribirlas en un archivo externo.

```
#include <iostream>           // define el flujo cout
#include <fstream>            // define la clase ofstream
using namespace std;

#include <iostream>
#include <stdlib.h>           // define la función exit()
using namespace std;

main( )
{
  ofstream archsal( "notas.dat", ios::out);
  if (!archsal) {
    cerr << "Error: no se puede abrir archivo de salida" << endl;
    exit(1);
  }
  char exp[7], nombre[20];
  int nota;
  cout << "\t1:";
  int n = 1;
  while (cin >> nombre >> exp >> nota) {
    archsal << nombre << " " << exp << " " << nota << endl;
    cout << "\t" << ++n << ":";
  }
  archsal.close( );
}
```

Al ejecutar el programa se introducen los datos de alumnos siguientes:

```
1:Mackoy       951234  7
2:Carrigan     962146  8
3:Mortimer     991045  9
4:Mackena      981146  9
5:García       982045  5
6:Rodríguez    991122  1
7:López        961333  6
```

El archivo creado `notas.dat` es

```
Mackoy     951234  7
Carrigan   962146  8
Mortimer   991045  9
Mackena    981146  9
García     982045  5
Rodríguez  991122  1
López      961333  6
```

Ejercicio 33.7

Leer datos de un archivo de datos externo (`notas.dat`).

La clase `iostream` se define en el archivo de cabecera `<fstream>`. Esta clase debe ser instanciada para leer de un archivo externo.

```
#include <iostream>     // define el flujo cout
#include <fstream>      // define la clase ofstream
#include <stdlib>       // define la función exit( )
main( )
{
  ifstream archen ("notas.dat, ios::in);
  if (!archen) {       // archivo de entrada, archen
    cerr << "Error: no se puede abrir archivo de entrada" << endl;
    exit(1);
```

```
  }
  char exp[7], nombre[20];
  int nota, suma = 0, cuenta = 0;
  while (archen >> nombre >> exp >> nota) {
    suma += nota;
    ++cuenta;
  }
  archen.close( );
  cout << "La nota media es" << float(suma)/cuenta << endl;
}
```

Si se ejecuta el programa anterior la salida será:

```
La nota media es 6.42857
```

E/S binaria

Existen varios modos de escribir y leer datos en binario desde un archivo. Un método es utilizar las funciones miembro `put ( )` y `get ( )`. Otro método es utilizar las funciones miembro `read ( )` y `write ( )`.

Funciones miembro `get` y `put`

La función miembro **`get ( )`** de la clase `istream` lee el flujo de entrada byte a byte (carácter). Existen tres formatos de la función `get( )`:

```
istream & get (char& car);
```

Extrae un único carácter del flujo de entrada, incluyendo espacio en blanco y lo almacena en `car`. Devuelve el objeto `istream` al que se aplica.

Ejemplo 33.20

El siguiente programa muestra en pantalla el contenido de un archivo.

```
#include <iostream>
#include <fstream>

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

  if (argc != 2) {
    cout << "error en número de archivos\n";
    return 1;
  }
  ifstream ent(argv[1], ios::in | ios::binary);
  if (!ent)
  {
    cerr << "No se puede abrir el archivo" << endl;
    return -1;
  }
  while (ent)     // ent será 0 cuando se alcance eof
  {
    ent.get(c);
    cout << c;
  }

  ent.close( );
  return 0;
}
```

El bucle `while` se termina cuando `ent` alcanza el final del archivo (`eof`, *end of file*) cuyo valor será nulo.

Consejo

Existe un modo más abreviado de implementar el bucle de lectura, aunque menos legible.

```
while (ent.get(c))
 cout << c;
```

Función `put ( )`

La función miembro `put ( )` proporciona un método alternativo de sacar (escribir) un carácter en el flujo de salida. `put ( )` acepta un argumento de tipo `char` y devuelve el objeto de la clase `ostream` a la cual se invoca.

Sintaxis

```
ostream& put(char&c);
```

Ejemplo 33.21

Escribir una cadena de caracteres no ASCII en el archivo.

```
#include <iostream>
#include <fstream>
using namespace std;

int main ( )
{
  char *p = "Hola colegas \n\r\xff";
  of stream sal( "prueba", ios: :out | ios: :binary);
  if (!sal)
  {
    cerr << "No se puede abrir el archivo" << endl;
    return -1;
  }

  while (*p) sal.put(*p++);

  sal.close( );

  return 0;
}
```

Formato 2 de `get ( )`

El segundo formato de `get ( )` lee también un único carácter del flujo de entrada. La diferencia es que devuelve el valor en lugar del objeto `istream` al que se aplica. Devuelve un tipo `int` en lugar de `char` debido a que devuelve también la representación final de archivo (**`eof`**), que suele representarse como –1 para diferencia de los valores del conjunto de caracteres.

Se ha de comprobar si el valor devuelto es final de archivo, para lo cual se compara con la constante `EOF` definida en el archivo de cabecera `iostream`. La variable asignada para contener el valor devuelto por `get ( )` se debe declarar del tipo `int`, de modo que pueda contener valores carácter y `EOF`.

Ejemplo

```
#include <iostream>
using namespace std;

int main( )
{
  int car;
  while ((car = cin.get( )) != EOF)
    cout.put(car);

  return 0;
}
```

Funciones `read` y `write`

El segundo método de lectura y escritura de datos binarios consiste en utilizar las funciones miembro `read( )` y `write( )`.

Sintaxis de `write ( )`

```
ostream& write(const char* buf, int num);
ostream& write(const unsigned char* buf, int num);
ostream& write(const signed char* buf, int num);
```

La función `write ( )` inserta (escribe) el número especificado de bytes (*num*) en el flujo asociado desde el búfer al que apunta *buf* (*buf* es un puntero a un arreglo de caracteres y *num* es el número de caracteres a escribir).

Sintaxis de `read ( )`

```
istream &read(char* buf, int num);
istream &read(unsigned char* buf, int num);
istream &read(signed char* buf, int num);
```

La función `read ( )` lee *num* bytes del flujo asociado y los coloca en el búfer al que apunta *buf*. El parámetro es un puntero a un arreglo de caracteres y *num* especifica el máximo número de caracteres a leer. La función miembro extrae caracteres hasta que se alcanza el final del archivo.

Ejemplo 33.22

Escritura de un arreglo de enteros y lectura posterior.

```
#include <iostream>
#include <fstream>
using namespace std;

int main( )
{
  int n[6] = {15, 25, 35, 45, 50, 60};
  register int i;
  of stream salida ("prueba", ios::out | ios::binary);
  if( !salida)
  {
    cerr << "No se puede abrir el archivo" << endl;
```

```
    return -1;
  }
  salida.write ((unsigned char*) n, sizeof) (n));

  salida.close();

  // inicializa el array
  for (i = 0; i < 6; i++)
    n[i] = 0;

  ifstream entrada ("prueba", ios::in | ios::binary);
  entrada.read ((unsigned char*) n, sizeof) (n));

  // presenta valores leídos del archivo
  for (i = 0; i < 6; i++)
    cout << n[i] << " ";
  entrada.close( );
  return 0;
}
```

Caracteres leídos

Si se alcanza el final del archivo antes de que se haya leído el número de caracteres especificado, la función `read ( )` se interrumpe, y el búfer contiene tantos caracteres como están disponibles. Si se desea conocer el número de caracteres que se han leído, se puede utilizar la función miembro `gcount ( )`, que devuelve el número de caracteres leídos en la última operación de entrada binaria.

Sintaxis

```
int gcount( );
```

Detección de EOF

El final del archivo se puede detectar mediante la función miembro `eof()`, que devuelve un valor distinto de cero cuando se alcanza el final de archivo; en caso distinto devuelve 0.

Sintaxis

```
int eof( );
```

Acceso aleatorio

En el sistema de E/S de C++ se pueden realizar accesos aleatorios mediante las funciones `seekg ( )` y `seekp( )`. Este tipo de acceso es usual en trabajos de bases de datos que permiten seleccionar y elegir registros especificados en archivos. Estas tareas se pueden realizar con las dos funciones miembro sobrecargadas heredadas de la clase `istream`.

El acceso a archivos aleatorios se describe generalmente en términos de un *puntero de archivo* (no confundir con punteros C++). Un puntero de archivo es un índice o apuntador que apunta a una posición dada o a la *posición actual* de un archivo entre el principio y el final de éste. La posición actual es el punto en el que comienza el siguiente acceso al archivo. Dado que el acceso al archivo se puede clasificar en términos de entrada y salida, la posición actual se puede referenciar como posición actual "obtener" (*current get position*) y una posición actual de "poner" (*current put position*). Las posiciones actuales *get* y *put* ayudan a introducir las funciones miembro asociadas de las clases `istream` y `ostream` para realizar *acceso aleatorio* o no secuencial.

El sistema de E/S de C++ maneja dos punteros asociados con el archivo. Uno es el puntero *get*, que especifica la posición del archivo en el que se producirá la siguiente operación de entrada. El otro es el puntero *put*, que especifica la posición del archivo en el que se producirá la siguiente operación

de salida. Cada vez que se realiza una operación de entrada o salida, el puntero correspondiente se avanza automáticamente. Sin embargo, cuando se utilizan las funciones `seekg ( )` y `seekp ( )` se puede acceder al archivo de modo aleatorio, no secuencial.

Las funciones de acceso aleatorio a archivos `seekg ( )` y `tellg ( )` tienen la sintaxis siguiente:

```
istream& seekg (streampos pos);
istream& seekg (streamoff desp, seek_dir dir);
streampos tellg( );
```

La versión de un argumento de `seekg ( )` mueve la posición actual *get* a una posición absoluta *pos*, en el flujo de entrada. El argumento *pos* es de tipo `streampos`, que es, simplemente, un `long typedef` definido en el archivo de cabecera *iostream*.

```
typedef long streampos;
```

La versión de dos argumentos de `seekg ( )` mueve *desp* bytes relativos a la posición actual *get* en la dirección especificada por *dir*. Al igual que `streampos`, `streamoff` está definido en el archivo *iostream*.

```
typedef long streamoff;
```

El tipo de `dir` es un tipo enumerado `seek_dir` y está declarado como miembro de la clase `ios`:

```
enum seek_dir
{
  beg,  // principio del archivo
  cur,  // posición actual
  end   // final del archivo
};
```

La función `seekg ( )` desplaza el puntero *get* del archivo asociado un número de bytes especificado en el desplazamiento *desp*, desde el origen *dir* que tomará uno de los tres valores del tipo `enum seek_dir`.

La función miembro `tellg( )` devuelve la posición actual del flujo de entrada. Su prototipo es:

```
streampos tellg( ):
```

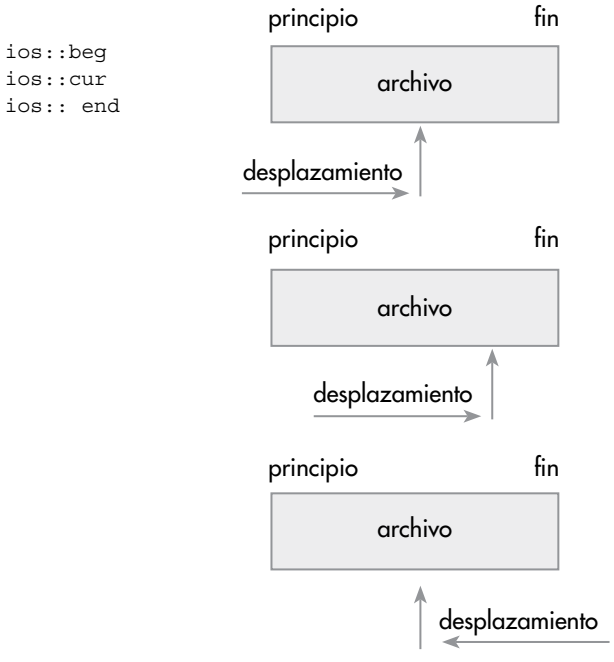


Figura 33.6 Posiciones de desplazamiento.

Ejemplo

```
// uso de las funciones miembro seekg( ) y tellg( )
#include <fstream>
using namespace std;

void main( )
{
  char nombre _arch[ ] = "CGP.CPP";
  ifstream arch_en (nombre _arch);
  if (!arch_en)
    cerr << "no se puede abrir el archivo\""
         << nombre _arch << "\" " << endl;

  // determinar la longitud del archivo
  streampos inicio = arch_en.seekg(0, ios::beg).tellg( );
  streampos fin = arch_en.seekg(0, ios::end) .tellg( );
  // obtener un desplazamiento
  streamoff desplazamiento;
  cout << "introducir un desplazamiento (+) desde el principio
          del archivo |""
       << nombre _arch << " \"" << endl
       << "cuya longitud tenga una longitud de archivo"
       << "(fin-inicio) << "caracteres:";
  cin >> desplazamiento;

  // ir a desplazamiento
  arch_en.seekg (desplazamiento, ios::beg);
  char car;
  while (arch_en)
  {
    arch_en.get(car);
    cout << car;
  }
  // cerrar
  arch_en.close( );
}
```

De igual modo, la clase `ostream` tiene dos funciones miembro: **`seekp ( )`** y **`tellp( ).`** Su sintaxis es:

```
ostream& seekp(streampos pos);
ostream& seekp(streamoff desp, seek_dir dir);
streampos tellp( );
```

La versión de un argumento de `seekp ( )` mueve la posición actual *put* a una posición absoluta, `pos`, en el flujo de salida. La versión de dos argumentos de `seekp ( )` mueve *desp* bytes relativos a la posición *put* actual en la dirección especificada por *dir.* La función miembro `tellp ( )` devuelve la posición actual del flujo de salida y devuelve un valor de −1 si ocurre un error.

Ejercicio 33.8

El siguiente programa permite especificar un nombre de archivo en la línea de órdenes, seguido del número del byte que se desea modificar en el archivo. Luego se escribe una `'T'` en la posición especificada.

```
#include <iostream>
#include <fstream>
#include <stdlib>
using namespace std;

int main(int argc, char *argv[ ])
{
   if(argc != 3) {
     cout << "Usar: CAMBIA <nombre_archivo> <byte>\n";
     return -1;
   }
   fstream sal(argv[1], ios::in | ios::out | ios::binary);
   if (!sal) {
     cerr << "No se puede abrir el archivo\n";
     return -1;
   }

   sal.seekp(atoi(argv[2]), ios::beg);

   sal.put('T');
   sal.close( );

   return 0;
}
```

Resumen

En este capítulo se ha aprendido a manejar las operaciones básicas de entrada y salida. Asimismo, se ha aprendido a utilizar los manipuladores para ajustar opciones de formatos para seleccionar entradas y salidas. Las características típicas de E/S son:

- Cuatro objetos de flujos se crean en cada programa: `cout`, `cin`, `cerr` y `clog`.
- La salida a pantalla utiliza el identificador de flujo `cout` y el operador de inserción.
  ```
  cout << valor;
  ```
- La entrada del teclado se realiza mediante el identificador de flujo `cin` y el operador de inserción `>>`.
  ```
  cin >> variable;
  ```
- La función miembro **`get ( )`** se utiliza para leer caracteres individuales desde el teclado y se puede utilizar **`put ( )`** para sacar caracteres.
- Cuando se necesita leer una línea completa de entrada, se debe utilizar la función miembro `getline( )`.
- Los manipuladores proporcionan un medio para modificar el flujo de entrada o salida. Un ejemplo típico, muy utilizado, es `endl` que sirve para insertar una nueva línea o retorno de carro.
  ```
  cout << setw(i)
  cout << hex; cout << oct
  cout << setprecision(i)
  cout << setiosflags(indicador)
  cout << resetiosflags(indicador)
  ```
- Algunas funciones miembro utilizadas en el capítulo son:
  ```
  cout.write(cadena, int i)
  cout.put(char car);
  cout.get( [char c]);
  cin.get(cadena, int i[, char term]);
  cin.getline(cadena, int [, char term]);
  ```

Un archivo de datos es una secuencia de bits almacenados en algún dispositivo de almacenamiento externo (cinta, disco, disquete, CD-ROM). Excepto en circunstancias extremas, los archivos

externos son permanentes: se pueden crear y guardar un día y recuperarlos posteriormente del disco en su computadora.

Se puede crear un archivo de datos externo utilizando un editor de texto de igual forma que se crea un archivo de programa y también crearse durante la ejecución de un programa. Un archivo de datos se puede leer muchas veces. De igual forma se puede instruir a un programa para que ejecute su salida en un archivo de disco en vez de visualizarlo en la pantalla de su monitor.

La escritura de programas que manipulan archivos de disco externos se complica con el hecho de que están implicados dos nombres diferentes: el *nombre del archivo externo*, que aparece en el directorio del disco en el que reside el archivo (nombre por el que conoce el sistema operativo al archivo), y el *nombre del objeto flujo*, que es el *nombre interno* por el que el programa accede al archivo. En C++, la conexión entre estos dos nombres se realiza mediante el uso de una función especial, denominada `open`.

Para abrir y cerrar archivos se crean objetos `if stream` y `of stream`. Para comenzar a escribir en un archivo, se debe crear primero un objeto `of stream` y, a continuación, asociar ese objeto con un archivo específico en su disco. Para utilizar objetos `of stream`, debe asegurarse de incluir el archivo `fstream` en su programa.

Nota: `fstream` *incluye* `iostream`, *por ello no es necesario incluir explícitamente* `iostream`.

Los archivos necesitan ser abiertos y cerrados. Estas operaciones se realizan con las funciones `open ( )` y `close ( )`. Las funciones implicadas en las operaciones de lectura y escritura son: `get/put` y `read/write` así como `seekp` y `seekg`.

Ejercicios

33.1. ¿Cuáles son los operadores de inserción y extracción? ¿Cuál es su misión?

33.2. Explicar las diferencias de los formatos de `cin.get ( )` y `cin.getline( )`.

33.3. Escribir un programa que escriba en los cuatro objetos `iostream` estándar: `cin`, `cout`, `cerr` y `clog`.

33.4. Escribir un programa que solicite al usuario introducir su nombre completo y, a continuación, que se visualice en pantalla.

33.5. Escribir un programa que utilice las funciones `setf( )`, `fill( )` y `width( )` que produzca la siguiente salida formateada:

Capítulo 5 Clases 300
Capítulo 6 Herencia 335
Capítulo 7 Flujos 355

33.6. Escribir el código para cada uno de los siguientes casos:

a) Imprimir el entero 12345 en un campo de 12 dígitos justificados a izquierda.
b) Imprimir 3.14159 en un campo de 12 dígitos con ceros precedentes.
c) Leer un entero en decimal e imprimirlo en octal.
d) Leer un entero en hexadecimal e imprimirlo en decimal.

33.7. Escribir una función `set_width (int w)` que establezca el campo con `cout` a w `columnas`.

33.8. Escribir un código C++ que lea una línea de texto y haga eco de la línea con todas las letras mayúsculas suprimidas.

33.9. Abrir un archivo de texto, leerlo línea a línea en un arreglo de caracteres y escribirlo de nuevo en otro archivo.

33.10. Copiar un archivo, en modo binario, carácter a carácter.

33.11. Escribir un programa que lea un número arbitrario de enteros de un archivo que tiene el siguiente formato:

```
8
7
6
5
4
3
2
1
```

33.12. Escribir un programa que lea un texto de la terminal y almacene el texto en un nuevo archivo de texto con el nombre `miarch.txt`. El nuevo archivo debe tener la misma estructura de línea que el archivo de texto escrito desde la terminal. Además, todas las letras minúsculas deben ser traducidas a letras mayúsculas.

33.13. Se dispone de un archivo `telefono.txt`, con nombres y números de teléfonos en orden alfabético. Escribir un programa que añada una nueva persona. El programa debe leer un nombre y número de teléfono de la nueva persona desde la terminal y, a continuación, insertar esta información en el lugar correcto del archivo de modo que permanezca ordenado. Sugerencia: utilice un archivo temporal.

33.14. La información acerca de un conjunto de personas se ha almacenado en un archivo binario para propósitos estadísticos. La información almacenada de cada persona es el nombre, altura, tamaño del calzado, edad y estado civil. Con el objeto de procesar los datos, el sexo de cada persona tiene que conocerse, aunque esta información no se incluye en el archivo. Escribir un programa que lea el archivo y cree dos archivos nuevos, uno que contenga sólo mujeres y otro sólo hombres. Por cada persona del archivo, el programa debe pedir al operador si una persona es un hombre o una mujer.

33.15. Escribir un programa que permita crear un archivo inventario de los libros de una librería, así como calcular e imprimir el valor total del inventario. Los campos de cada libro deben ser, como mínimo, título, autor, ISBN, precio, cantidad, editorial.

Problemas

33.1. Se trata de declarar una cadena de caracteres para leer el nombre del archivo, y abrir el archivo en modo entrada. En caso de que no pueda abrirse se debe crear de escritura. Una observación importante es que siempre se ha de comprobar si la apertura del archivo ha sido realizada con éxito, puesto que es una operación que realiza el sistema operativo para el programa y queda fuera de nuestro control; esta operación se realiza comprobando el valor de good.

33.2. Una vez que se conoce el funcionamiento del operador de extracción `>>` para el objeto `cin`, es muy sencillo su uso para cualquier objeto de la clase `fstream`, ya que su funcionamiento es el mismo. El archivo abierto contiene números enteros, pero al ser un archivo de texto esos números están almacenados no de forma binaria sino como una cadena de caracteres que representan los dígitos y el signo del número en forma de secuencia de sus códigos ASCII bina-

rios. Esto no quiere decir que haya que leer línea a línea y en cada una de ellas convertir las secuencias de códigos de caracteres a los números enteros en binario correspondiente, para almacenarlos así en la memoria. Este trabajo es el que realiza el operador de extracción >> cuando tiene una variable de tipo entero en su parte derecha. Es la misma operación de conversión que realiza el operador de extracción cuando lee secuencias de códigos de teclas desde la entrada estándar. Escribir un programa que utilice dos contadores para contar los positivos y negativos y una variable para leer el dato del archivo. El programa visualizará además el archivo.