La biblioteca estándar de E/S (*IOstreams*)

Inmaculada Medina, Francisco Palomo, Gerardo Aburruzaga

Abril de 2004

Índice

1. La biblioteca estándar de E/S

Hasta ahora se ha utilizado solamente E/S elemental: lectura de tipos básicos desde la entrada estándar y su escritura por la salida estándar, mediante los operadores de inserción y extracción. Ahora se tratará todo esto en más detalle.

En C++, como en C, no hay nada incorporado en el lenguaje sobre E/S. Todas las funciones de E/S deben ser definidas. Como la E/S es muy dependiente

del sistema, se espera que en cada uno el fabricante suministre las funciones más o menos básicas que permitan trabajar con ficheros, consola, terminal, etc. Estas funciones suelen agruparse para formar bibliotecas.

Afortunadamente la biblioteca de E/S de UNIX, por ser la más sencilla, fue adaptada por el comité ANSI de C y adoptada como estándar. En ella se definía el concepto de *flujo* de datos: una corriente de bytes que actúa como fuente o destino de datos según sea de entrada o de salida.

Todas las operaciones corrientes se pueden hacer mediante los flujos; por muy complicado que sea el soporte que proporcione el sistema operativo para los distintos periféricos, es la biblioteca la que se encarga de las transformaciones. El flujo actúa como una interfaz entre el programador y el sistema operativo. Cuando se desee más control o rapidez, habrá que utilizar las funciones proporcionadas por el sistema.

En C++, heredero de C, puede seguirse empleando la biblioteca de C de E/S: no hay más que incluir la ya conocida cabecera `<cstdio>` y utilizar las conocidas funciones. Sin embargo, pocos son los que, proponiéndose programar en serio en C++, utilizan esta biblioteca.

La biblioteca de E/S para C++ sigue conservando el concepto de flujo, como en C. ¿Cuáles son las ventajas de esta biblioteca sobre la de C?

- Mejor comprobación de los tipos de los datos de E/S. En las famosas funciones *printf()* y *scanf()* no hay comprobación de tipos, como debe saber: sólo el primer argumento es conocido; el tipo y número de los siguientes, si los hay, se basan en la cadena de formato[1].
- Tratamiento uniforme de todos los tipos de datos. Esto incluye, y es muy importante, los definidos por el usuario.
- Capacidad de extensión. Se puede crear un nuevo flujo de datos, se pueden sobrecargar los operadores de E/S, etc.

Para utilizar la biblioteca de E/S de C++ hay que incluir, fundamentalmente, el fichero de cabecera `<iostream>`, como se ha hecho hasta ahora. Se verá que a veces habrá que incluir también alguna otra cabecera, dependiendo de lo que se vaya a hacer.

No se pretende dar una descripción muy detallada de la biblioteca; para ello consúltese la bibliografía. Se expondrá aquí un resumen de sus capacidades, recalcando los aspectos más útiles.

En la figura 1 se muestra un esquema con la jerarquía de clases de los flujos; algunas de estas clases se verá a continuación.

2. Flujos de salida

En `<iostream>` se define la clase *ostream*, una especialización para una modalidad concreta de carácter de la plantilla *basic_ostream*.

```
typedef basic_ostream<char> ostream;
```

[1] Aunque algunos compiladores analizan la cadena de formato y así comprobar el resto de los argumentos.

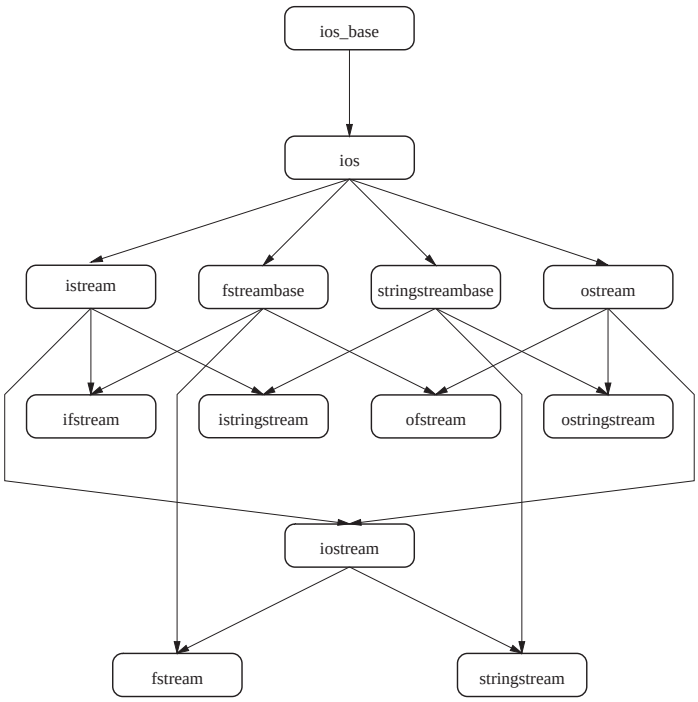

Figura 1: Jerarquía de clases de flujos

Esta clase plantilla *basic_ostream*, derivada de la clase *basic_ios* presente en `<ios>`, y sus operaciones asociadas de salida se definen en el espacio de nombres `std` y las presenta `<ostream>`, que contiene los componentes de `<iostream>` relacionados con la salida.

Algunos miembros públicos de esta clase son:

```
ostream& operator <<(bool);            // lógico
ostream& operator <<(int);             // entero
ostream& operator <<(double);          // real en doble precisión
ostream& operator <<(const void*);     // puntero, dirección
ostream& put(char c);                  // carácter
ostream& flush();                      // vacía el búfer
ostream& write(const char* s, streamsize n);
                                       // cadena de bajo nivel
                                       // (hasta n caracteres)
```

streamsize es un tipo entero con signo que se emplea para representar el número de caracteres transferidos en una operación de E/S y el tamaño de los búferes de E/S.

La salida se envía a un objeto de la clase *ostream*, lo que permite por ejemplo concatenar operaciones, como se ha hecho ya con el operador `«`. Existen algunos objetos predefinidos que ya se conocen:

cout	Salida estándar normal
cerr	Salida estándar de errores
clog	Salida estándar de errores, pero el búfer se vacía sólo cuando está lleno

Ejemplo:

```
cout.put('A');                // imprime A
const char* str = "ABCDEFGHI";
cout.write(str + 2, 3);       // imprime CDE
cout.flush();                 // vacía el búfer de salida
```

Hay que señalar que un valor *bool* se escribirá por omisión en la salida como 0 o 1. Si se desea obtener `true` o `false` se puede fijar el indicador de formato *boolalpha*[2].

Ejemplo:

boolean.cpp

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

int main()
{
  cout « true « ' ' « false « '\n'
       « boolalpha       // para usar representación simbólica
       « true « ' ' « false « endl;
}
```

Esto imprimiría:

```
1 0
true false
```

Más precisamente, *boolalpha* garantiza que obtendremos una representación de los valores *bool* dependiente de la *localización*.

2.1. El operador de inserción «

El operador de desplazamiento de *bits* hacia la izquierda, «, como ya se sabe, está sobrecargado para trabajar con los tipos básicos. Su asociatividad y precedencia es la misma que la del operador incorporado en el lenguaje, lo cual es conveniente la mayoría de las veces. Por ejemplo,

```
cout << "x = " << x << "\n";
```

equivale a

```
((cout.operator<<("x = ")).operator<<(x)).operator("\n");
```

Además, la precedencia de « es lo suficientemente baja como para permitir expresiones aritméticas como operandos sin utilizar paréntesis.

[2]Esto en algunos compiladores todavía no está incluido

Ejemplo:

```
cout << "a * b + c = " << a * b + c << '\n';
```

Aunque, a veces la precedencia da algún problema:

```
cout << a + b << " es una suma\n";               // bien
cout << a & b << " es un problema\n";            // ERROR
cout << (a & b) << " es un problema resuelto\n"; // bien
```

Se puede utilizar el operador de desplazamiento a la izquierda en una instrucción de salida pero, naturalmente, debe aparecer entre paréntesis:

```
cout << "a << b = " << (a << b) << '\n';
```

2.2. Salida formateada

2.2.1. Campos de salida

A menudo se desea rellenar con texto un espacio específico de una línea de salida. Se quieren utilizar n caracteres y no menos (y más solamente si no encaja el texto). Para hacerlo especificamos una anchura de campo y un carácter de relleno:

```
class ios_base {
public:
  // ...
  streamsize width() const;  // obtener anchura de campo
  streamsize width(streamsize wide);
                             // establecer anchura de campo
  // ...
};

class ios: public ios_base {
public:
  // ...
  char fill() const;         // obtener carácter de relleno
  char fill(char c);         // establecer carácter de relleno
  // ...
};
```

La función *width()* especifica el número mínimo de caracteres a utilizar para la siguiente operación de salida « de un valor numérico (*bool*, cadena de bajo nivel, carácter, puntero, *string*).

Ejemplo:

campo-salida.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    cout.width(4);
7    cout « 10;
8  }
```

imprimiría 10 precedido de dos espacios.

El carácter de relleno se puede especificar con la función *fill()*.

Ejemplo:

relleno-salida.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    cout.width(4);
7    cout.fill('#');
8    cout « 10;
9  }
```

El carácter de relleno por omisión es el de espacio y la anchura por omisión es 0, que significa: tantos caracteres como sean necesarios.

El tamaño de campo se puede reestablecer a su valor por omisión de la siguiente forma:

```
cout.width(0);
```

La función *width()* establece el número mínimo de caracteres. Si se proporcionan más caracteres se imprimirán todos.

Ejemplo:

```
cout.width(4);
cout << 12345;
```

imprimiría 12345 y no sólo 1234.

Una llamada a `width(n)` afecta sólo a la operación de salida `«` inmediatamente posterior.

Ejemplo:

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;
```

Esto produce `##12:13`, en lugar de `##12:##13`.

Los manipuladores que veremos a continuación proporcionan una forma más elegante de especificar la anchura de un campo de salida.

2.2.2. Ajuste de campo

El ajuste de los caracteres dentro de un campo se puede controlar mediante llamadas a *setf()*:

```
cout.setf(ios_base::left, ios_base::adjustfield);     // izquierda
cout.setf(ios_base::right, ios_base::adjustfield);    // derecha
cout.setf(ios_base::internal, ios_base::adjustfield); // interno
```

Esto establece el ajuste de la salida en un campo de salida definido por `ios_base::width()` sin efectos colaterales sobre otras partes del estado del flujo.

Ejemplo:

ajuste-salida.cpp

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

int main()
{
  cout.fill('#');
  cout « '(';
  cout.width(4);
  cout « -10 « "), (";

  cout.width(4);
  cout.setf(ios_base::left, ios_base::adjustfield);
  cout « -10 « "), (";

  cout.width(4);
  cout.setf(ios_base::internal, ios_base::adjustfield);
  cout « -10 « ')' « endl;
}
```

Esto imprimiría: `(#-10), (-10#), (-#10)`

2.2.3. Manipuladores

El operador « produce el mínimo número de caracteres; por ejemplo:

```
cout << 2.0;                          // imprime 2 y no 2.0

int i = 6, j = 7;
cout << i << j;                       // imprime 67
cout << i << " " << j;                // mejor: imprime 6 7
```

El método consistente en imprimir espacios, tabuladores y nuevas líneas donde haga falta puede ser suficiente en algunos casos, pero a veces se echan de menos algunas capacidades de formato como las que tenía *printf()*. No obstante, en la biblioteca de E/S en C++ se dispone de los *manipuladores*, que son valores o funciones que, sin imprimir nada visible, producen algún efecto especial en el flujo en el que operan. Ya se ha visto uno, *endl*, que escribía `'\n'` y vaciaba el

Manipulador	Descripción
`endl`	Nueva línea, vacía el búfer de salida
`ends`	Carácter nulo (`'\0'`, terminador de cadena)
`flush`	Vacía el búfer de salida
`dec`	Números en base decimal
`oct`	Números en base octal
`hex`	Números en base hexadecimal
`ws`	Saltar espacios en blanco
`uppercase`	Pasar a mayúsculas
`nouppercase`	Pasar a minúsculas

Cuadro 1: Manipuladores en `<ios>`, `<ostream>` e `<iostream>`

Manipulador	Descripción
`setw(int)`	Anchura de campo
`setfill(int)`	Carácter de relleno
`setbase(int)`	Base de numeración
`setprecision(int)`	Precisión de reales
`setiosflags(ios_base::fmtflags)`	Establecer indicadores
`resetiosflags(ios_base::fmtflags)`	Borrar indicadores

Cuadro 2: Manipuladores en `<iomanip>`

búfer de salida. De esta forma, ya no hay que manejar el estado de un flujo en términos de indicadores, sino a través de los manipuladores.

Los manipuladores más normales se recogen en la tabla 1. Otros (justo los que actúan como funciones, y reciben parámetros por tanto) requieren la inclusión de otra cabecera estándar: `<iomanip>`. Éstos se recogen en la tabla 2.

Los *bits* de formato de los manipuladores (*re*)*setiosflags()* se pueden nombrar con las constantes enumeradas de la clase *ios* que se muestran en la tabla 3 (aparecen sus nombres, estado de activación predeterminado y significado). Para poner varias de ellas se empleará el operador de *bits* OR (`bit_or` o `|`); por ejemplo:

```
cout << setiosflags(ios_base::left |
                    ios_base::hex  |
                    ios_base::uppercase) << ...
```

Ejemplo:

Como pequeño ejemplo, el siguiente programa manipula enteros en diferentes bases. Observe que estos manipuladores también valen para la entrada.

bases.cpp

```
1  // Utilizando varias bases en la E/S de enteros
2
3  #include <iostream>   // <iomanip> no hace falta
4  using namespace std;
5
6  int main()
```

Nombre	**Estado**	**Significado**
`skipws`	Sí	Saltar blancos en la entrada
`left`	No	Salida justificada a la izquierda
`right`	No	Salida justificada a la derecha
`internal`	No	Signo justificado a la izda., núm. a la derecha
`dec`	Sí	Base decimal. Equivale al manipulador `dec`
`oct`	No	Base octal. Equivale al manipulador `oct`
`hex`	No	Base hexadecimal. Equivale al manipulador `hex`
`showbase`	No	Emplear indicador de base: `0x` para `hex` y `0` para `oct`
`showpoint`	No	Poner punto o coma decimal y ceros decimales en números reales aunque sea superfluo
`uppercase`	No	Emplear mayúsculas para las letras de los números hexadecimales y el indicador de exponente de 10
`showpos`	No	Mostrar siempre el signo, aunque sea positivo
`scientific`	No	Forzar notación científica (exponencial) en números reales
`fixed`	No	Forzar notación fija en números reales
`unitbuf`	No	Vaciar todos los búferes tras la salida
`stdio`	No	Vaciar los búferes de *cout* y *cerr* tras la salida

Cuadro 3: Nombres de los *bits* de formato

```
7  {
8    int i = 10, j = 16, k = 24;
9
10   cout « i « '\t' « j « '\t' « k « endl
11        « oct « i « '\t' « j « '\t' « k « endl
12        « hex « i « '\t' « j « '\t' « k « endl
13        « "Introduzca tres enteros;"
14        « " por ejemplo: 11 11 12a" « endl;
15   cin » i » hex » j » k;
16   cout « dec « i « '\t' « j « '\t' « k « endl;
17 }
```

La salida resultante para la entrada de ejemplo es:

```
10      16      24
12      20      30
a       10      18
Introduzca tres enteros; por ejemplo: 11 11 12a
11      17      298
```

2.3. Salida de tipos definidos por el usuario

El operador « se puede sobrecargar para insertar en un flujo cualquier tipo nuevo que se defina.

En vez de definir funciones miembro para mostrar los objetos de una clase, está más de acuerdo con la filosofía de C++ y de esta biblioteca el sobrecargar el

operador « para ello: los tipos definidios por el usuario deberían saber mostrarse como los demás. Para ello se empleará el siguiente esqueleto:

```
ostream& operator <<(ostream& os, const T& obj)
{
  // lógica especial para el objeto de tipo T
  return os; // devolución del flujo de salida
}
```

El primer argumento es una referencia a un flujo de salida; debido a esto, este operador debe ser definido como una función no miembro, fuera de la clase. Si debe tener acceso a sus miembros privados, como es común, debe ser declarado en ella como amigo. Devuelve el mismo flujo de salida como referencia para poder concatenar las operaciones.

Ejemplo:

Dada una clase para manejar números racionales veamos cómo se puede sobrecargar el operador de inserción.

```
class racional {
public:
  friend ostream& operator <<(ostream&, const racional&);
  // ...
private:
  int n, d;
};

ostream& operator <<(ostream& salida, const racional& r)
{
  salida << r.n;
  if (r.n != 0 && r.d != 1)
    salida << '/' << r.d;
  return salida;
}
```

3. Flujos de entrada

La entrada se maneja de forma similar a la salida. Como era de esperar el operador de desplazamiento de *bits* a la derecha se sobrecarga en la clase *istream* para los tipos básicos, y se llama entonces *extracción*. También se define un objeto de esta clase que representa al flujo de la entrada estándar: *cin*.

Análogamente a *basic_ostream*, *basic_istream* se define en `<istream>`, que contiene los componentes de `<iostream>` relacionados con la entrada.

Así, la clase *istream* se define como una especialización para una modalidad concreta de carácter de la plantilla *basic_istream*.

```
typedef basic_istream<char> istream;
```

Algunos miembros públicos de la clase *istream* son:

```
istream& operator >>(int& i);    // entero
istream& operator >>(float& f);  // real de simple precisión
istream& operator >>(double& x); // real de doble precisión
int_type get();                  // carácter, incluido blanco, y EOF
istream& get(char& c);           // carácter, incluido blanco
istream& get(char* s, streamsize n, char d = '\n');
                                 // cadena de caracteres
istream& getline(char* s, streamzize n, char d = '\n');
                                 // cadena de caracteres
istream& read(char* s, streamsize n);
                                 // cadena de caracteres
istream& putback(char c);        // devuelve a la entrada el cter.
int peek();                      // toma el siguiente carácter
                                 // sin extracción
int gcount();                    // da el número de caracteres
                                 // recién extraídos
```

Gracias a las referencias ya no hace falta emplear punteros como ocurría en la función *scanf()* de la biblioteca de C.

El método `get(char* s, streamsize n, char d = '\n')` lee como mucho $n-1$ caracteres y los guarda en la zona apuntada por s, hasta encontrar el fin de la entrada o el carácter d, que hace de delimitador y que, como se ve, si no se suministra es el carácter nueva-línea. Este carácter delimitador no se guarda, y se añade siempre al final un carácter nulo `'\0'`.

El método *getline()* funciona como el anterior, sólo que no deja el carácter delimitador en el flujo de entrada; como si lo hubiera leído.

El método *read()* introduce como mucho n caracteres en la zona apuntada por s, pero si antes encuentra el fin de la entrada lo considera como error, activando *failbit* (§7).

Ejemplo:

```
int x, y;
char c, s[80];
cin.get(c);            // coge un carácter
cin.get(s, 40);        // cadena de 40 cars. o hasta EOF
cin.get(s, 9, ':');    // cadena hasta 9 caracteres o hasta ':'
cin.getline(s, 60);    // como get, pero '\n' se lee y desecha
cin >> x >> y;         // lee x e y
cin >> s;              // lee una cadena, delimitada por blancos
```

Note que en el último ejemplo no se lee una línea, sino caracteres delimitados por blancos (espacio, tabulador, nueva línea, etc.).

3.1. El operador de extracción »

El operador » se salta los espacios en blanco, por tanto se podría leer una secuencia de enteros separados por espacios en blancos de la siguiente forma:

```
vector<int> v(10);

for (vector<int>::size_type i = 0; i < v.size() && cin; ++i)
  cin >> v[i];
```

También podemos leer una cadena de bajo nivel directamente:

```
char v[4];

cin >> v;
cout << "v = " << v << endl;
```

El operador » salta primero los espacios en blanco. A continuación lee a su operando vector hasta que encuentra un carácter de espacio en blanco o llega al final del fichero. Por último, termina la cadena con un 0. Esto ofrece claramente oportunidades de desbordamiento, por lo que habitualmente es mejor leer a un *string*:

```
string s;
getline(cin, s);
```

Sin embargo, se puede especificar un máximo para el número de caracteres que va a leer ». Así, `cin.width(n)` especifica que el » siguiente en *cin* va a leer como máximo $n - 1$ caracteres en un vector.

Ejemplo:

ancho.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    char v[4];
7
8    cin.width(4);
9    cin » v;
10   cout « "v = " « v « endl;
11 }
```

Esto leerá un máximo de tres caracteres en v y añadirá un 0 como terminador.

Especificar *width()* para un *istream* afecta sólo al » que sigue inmediatamente a un vector y no afecta a la lectura a otros tipos de variables.

Ahora se verá un programa completo que lee una secuencia de palabras de la entrada estándar y determina cuál es la mayor. El término *palabra* no hay que tomarlo al pie de la letra; sería correcto si sólo se suministraran al programa palabras separadas por blancos, sin signos de puntuación. Note la condición del bucle `while`; existe un operador definido en la clase que transforma *istream* a *void** (§7), y el puntero se transforma a *bool* en la condición automáticamente. Cuando se llega al final de la entrada la expresión da cero; o sea, falso.

pml.cpp

```
1  // Lee palabras de la entrada estándar,
2  // determina la más larga
3
4  #include <iostream>
```

```
5  #include <string>
6  using namespace std;
7
8  int main()
9  {
10   string palabra;
11   string palabra_mas_larga;
12   size_t longitud, max = 0, n = 0;  // estadísticas
13
14   while (cin » palabra) {
15     longitud = palabra.length();
16     ++n;
17     // Si es la más larga, guardarla
18     if (longitud > max) {
19       max = longitud;
20       palabra_mas_larga = palabra;
21     }
22   }
23   cout « "El número total de palabras leídas es  "
24        « n « endl
25        « "La palabra más larga tiene de longitud "
26        « max « endl
27        « "La palabra más larga ha resultado ser  "
28        « palabra_mas_larga « endl;
29 }
```

3.2. Entrada de caracteres

El operador » está pensado para leer objetos de un tipo y un formato esperados. Cuando esto no es deseable y queremos leer los caracteres como tales y luego examinarlos, utilizamos las funciones *get()* y *getline()*.

Los dos siguientes programas leen de la entrada estándar copiándola en la salida, como el conocido programa de Unix `cat` empleado sin parámetros. Se llama a la función *get()* con un parámetro, una referencia a carácter. Como devuelve el objeto del mismo flujo, se puede utilizar como condición del bucle `while`, por lo dicho anteriormente de la conversión de *istream* a *void**.

cat1.cpp

```
1  // Copia la entrada a la salida (estándares)
2  // Ejemplo de get(char&)
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9    char c;
10   while (cin.get(c))
11     cout.put(c);
12 }
```

En este otro se llama sin argumentos, y entonces se parece a *getchar()*: al llegar al final de la entrada devuelve la constante *EOF*, definida en `<iostream>`.

cat2.cpp

```
// Copia la entrada a la salida (estándares)
// Ejemplo de get()

#include <iostream>
using namespace std;

int main()
{
  int c;  // no char: para poder recibir EOF
  while ((c = cin.get()) != EOF)
    cout.put(char(c));
}
```

Por último, el siguiente programa, que lee un texto de la entrada estándar y lo imprime «en detalle», emplea *getline()*, *gcount()* y *write()*.

lineas.cpp

```
// Lee líneas de texto de la entrada estándar.
// Las copia en la salida indicando el número
// de caracteres de cada una; y, al final, la
// longitud de la más larga.

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

const int TAMBUF = 256;

int main()
{
  int cl  =  0;        // Cuántas líneas llevamos leídas
  int max = -1;        // Longitud de la mayor
  char cadena[TAMBUF]; // Para almacenar cada línea

  // Lee TAMBUF caracteres o hasta nueva-línea
  while (cin.getline(cadena, TAMBUF)) {
    // cuántos caracteres hemos leído de verdad
    int leidos = cin.gcount();
    // datos: número de línea, línea más larga...
    cl++;
    if (leidos > max)
      max = leidos;
    cout « "Línea número " « cl
         « "\tCaracteres leídos: " « leidos « endl;
    cout.write(cadena, leidos).put('\n').put('\n');
  }
  cout « "Total de líneas leídas: " « cl « endl
       « "Longitud de la línea más larga: " « max « endl;
}
```

3.3. Entrada de tipos definidos por el usuario

Como con el operador inserción, el de extracción se puede sobrecargar en una clase para que ésta pueda recibir valores de una manera natural. El esqueleto sería así:

```
istream& operator >>(istream& is, T& obj)
{
  // Lógica para tratar con el objeto de tipo T
  return is;  // devolución del flujo de entrada
}
```

Ejemplo:

Retomando el ejemplo de los números racionales, ésta podría ser una forma muy abreviada y simplificada de introducción de un racional.

```
inline istream& operator >>(istream& is, racional& r)
{
  is >> r.n;
  char c = 0;
  is >> c;
  if (c == '/')
    is >> r.d;
  else
    is.putback(c).clear(ios_base::badbit);
  return is;
}
```

4. Funciones virtuales de E/S

Los miembros de *iostream* no son `virtual`. Las operaciones de salida que se pueden definir no son miembros de ninguna clase, por lo que tampoco pueden ser `virtual`. La razón es lograr un mayor rendimiento en operaciones sencillas como poner un carácter en un búfer.

Sin embargo, a veces se quiere mandar a una salida un objeto y se quiere que se elija en tiempo de ejecución la definición del operador adecuada dentro de la jerarquía de clases. Esto no se puede hacer utilizándolo directamente. En su lugar se puede proporcionar en la clase base una función virtual de salida.

Ejemplo:

salida-virtual.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  class Base {
5  public:
6    // ...
7    virtual ostream& poner(ostream& s) const = 0;
8  };
```

```
9
10  class Derivada: public Base {
11  public:
12    // ...
13    ostream& poner(ostream& s) const { return s « d « endl; }
14  private:
15    int d;
16  };
17
18  ostream& operator «(ostream& s, const Base& b)
19  {
20    return b.poner(s);
21  }
22
23  int main()
24  {
25    Base *b;
26    Derivada d;
27
28    b = &d;
29
30    cout « *b « d;
31  }
32
```

5. Flujos de fichero

Para la E/S de ficheros hay que incluir otra cabecera: `<fstream>`. En ella se definen las clases *ifstream* y *ofstream* para la creación y manipulación de ficheros de entrada y de salida respectivamente.

5.1. Apertura

Como ya se sabe, la *apertura* de un fichero es la operación que lo hace accesible al programa; se puede decir que se conecta un fichero (con posible creación si no existía, o truncamiento y vaciado) con un flujo que se manipula en el programa.

En la biblioteca de C existía una función para este propósito: *fopen()*. En la de C++ la apertura puede hacerse de dos maneras. De una o de otra, lo primero es definir un objeto del tipo *ifstream* u *ofstream*. Constructores:

```
ifstream();
ifstream(const char* camino,
         openmode modo = ios_base::in | ios_base::trunc);
ofstream();
ofstream(const char* camino,
         openmode modo = ios_base::out | ios_base::trunc);
```

Si se ha empleado el constructor sin parámetros, entonces aún no se ha abierto el fichero, sólo se ha creado el objeto de tipo flujo; se tendrá que abrir luego mediante los métodos siguientes:

Modo	Significado
`ios_base::in`	Lectura (entrada)
`ios_base::out`	Escritura (salida)
`ios_base::ate`	Colocación al final del fichero
`ios_base::app`	Modo de anexado; toda escritura ocurre al final del fichero
`ios_base::trunc`	Borra el contenido del fichero al abrir si ya existe
`ios_base::bin`	El fichero es binario; en UNIX no tiene efecto

Cuadro 4: Modos de apertura de ficheros

```
void
ifstream::open(const char* camino,
               openmode modo = ios_base::in | ios_base::trunc);
void
ofstream::open(const char* camino,
               openmode modo = ios_base::out | ios_base::trunc);
```

Tanto para las funciones *open()* como para los constructores con dos parámetros, sus significados son como sigue:

camino El nombre o camino del fichero que se quiere abrir. Los detalles dependen del sistema operativo, evidentemente. Si debe existir previamente o no, depende del modo de apertura.

modo El modo de apertura. Es un parámetro opcional. Los valores permitidos son constantes enumeradas de tipo *ios_base::open_mode*, como se muestran en la tabla 4. Se pueden especificar varios, aplicando el operador de bits '`|`'.

Si se define un objeto de la clase *fstream* el constructor hace que la apertura sea tanto para entrada como para salida; el parámetro *modo* debe ser suministrado por nosotros.

5.2. Cierre

El cierre de un fichero implica la desconexión del fichero físico con el flujo que se utiliza en el programa: el fichero ya no está accesible, los búferes se han vaciado. Existen tres formas (normales) de cerrar un fichero:

- Mediante el destructor de la clase asociada:

  ```
  ~ifstream();
  ~ofstream();
  ~fstream();
  ```

- Mediante el método *close()*.

- Mediante la función *exit()* de la biblioteca estándar de C (declarada en la cabecera `<cstdlib>`), que provoca la terminación normal del programa, o la sentencia `return` en la función *main()* o en definitiva una salida normal del programa (se llamaría al destructor).

Después de todo esto se van a ver algunos ejemplos.

EJEMPLO:

Esto es un pequeño programa que recibe dos argumentos que son nombres de ficheros. Copia el primero en el segundo pero separando cada línea por una en blanco.

dblspc.cpp

```
1  // Copia un fichero a doble espacio
2  // Modo de empleo: dblspc <fichero-entrada> <fichero-salida>
3  // 'fichero-entrada' debe existir y poderse leer
4  // 'fichero-salida' debe poder ser modificado si existe
5
6  #include <fstream>  // ya incluye <iostream>
7  #include <cstdlib>  // para exit() y sus macros
8  using namespace std;
9
10 void doblespacia(ifstream& entrada, ofstream& salida);
11
12 int main(int argc, char* argv[])
13 {
14   if (argc != 3) {
15     cerr « "Modo de empleo: " « argv[0]
16          « " fichen fichsal\n";
17     exit(EXIT_FAILURE);
18   }
19
20   ifstream flujoent(argv[1]);
21   ofstream flujosal(argv[2]);
22   if (!flujoent) {
23     cerr « "No puedo abrir " « argv[1] « endl;
24     exit(EXIT_FAILURE);
25   }
26   if (!flujosal) {
27     cerr « "No puedo abrir " « argv[2] « endl;
28     exit(EXIT_FAILURE);
29   }
30   doblespacia(flujoent, flujosal);
31   return EXIT_SUCCESS;
32 }
33
34 void doblespacia(ifstream& e, ofstream& s)
35 {
36   char c;
37
38   while (e.get(c)) {
39     s.put(c);
40     if (c == '\n')
41       s.put(c);
42   }
43 }
```

EJEMPLO:

Este otro programa imprime un fichero completo en la salida. Para ello hace uso del método *rdbuf()* que devuelve un puntero al búfer interno del flujo.

`ios_base::beg`	relativo al principio
`ios_base::cur`	relativo a la posición actual
`ios_base::end`	relativo al final

Cuadro 5: Valores de la enumeración *seek_dir*

rdbuf.cpp

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main()
6  {
7    ifstream motd("/etc/motd");
8
9    cout « motd.rdbuf();
10 }
```

De forma similar se puede hacer un programa que copia la entrada a la salida.

entrada-salida.cpp

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main()
6  {
7    cout « cin.rdbuf();
8  }
```

5.3. Acceso directo

Si un flujo está conectado a un fichero en un dispositivo que permite acceso directo, es posible moverse a un punto determinado del fichero. Para ello se dispone de algunos métodos.

Los dos siguientes métodos permiten situarnos en la posición especificada por *desplto* a partir de lo indicado por el segundo parámetro, *orig*. Este segundo parámetro, el *origen* del *desplazamiento*, es opcional y puede tomar alguno de los valores mostrados en la tabla 5.

```
ostream& seekp(off_type desplto, ios_base::seekdir orig);
istream& seekg(off_type desplto, ios_base::seekdir orig);
```

Los dos siguientes permiten situarnos en una posición absoluta.

```
ostream& seekp(pos_type);
istream& seekg(pos_type);
```

Y los siguientes devuelven la posición actual en el flujo de entrada o salida respectivamente.

```
pos_type tellp();
pos_type tellg();
```

La última letra de estas funciones es *g*, por *get*, coger de la entrada, o *p*, por *put*, poner en la salida.

6. Flujos de cadena

En la biblioteca estándar de C existen las funciones *sprintf()* y *sscanf()* que leen y escriben no en un flujo sino en una cadena de caracteres que tratan como tal. De esta forma, se puede hacer conversiones de datos numéricos a o desde cadenas, aprovechando las capacidades de formato de dichas funciones, y guardar los resultados para su posterior procesado. La biblioteca *IOStreams* de C++ también permite esto.

Así, se puede conectar un flujo a un *string*. Es decir, podemos leer y escribir un *string* usando los recursos de formato que proporcionan los flujos. Estos flujos de cadena se denominan *stringstream* y se definen en `<sstream>`.

Por ejemplo, se puede utilizar un *ostringstream* para formatear objetos *string*.

```
string error(int n, const& string s)
{
  ostringstream o;

  o << "Error " << n << ": " << s << endl;
  return o.str();
}
```

No es necesario comprobar el desbordamiento porque `o` se expande a medida de las necesidades.

Se puede proporcionar un valor inicial para un *ostringstream*, por lo que podríamos haber escrito de manera equivalente:

```
string error(int n, const string& s)
{
  ostringstream o("Error");

  o << n << ": " << s << endl;
  return o.str();
}
```

Un *istringstream* es un flujo de entrada que lee de un *string*:

```
void palabras(const string& s)
{
  istringstream i(s);
  string w;

  while (i >> w)       // lee una "palabra"
    cout << w << endl;
}
```

Ejemplo:

El siguiente programa lee de un fichero de configuración, `colores.ini`, nombres de colores y sus componentes RGB (rojo, verde, azul), y los muestra por la salida estándar formateados.

colores.cpp

```
1  // Lee del fichero colores.ini líneas en el formato
2  // color R G B
3  // y las imprime en la salida estándar con el formato
4  // color:        (R,G,B)
5  // con los números de los componentes en hexadecimal.
6
7  #include <iostream>
8  #include <fstream>
9  #include <iomanip>
10 #include <sstream>
11 #include <string>
12 using namespace std;
13
14 int main()
15 {
16   string color;     // nombre de color
17   unsigned int rojo, verde, azul;
18
19   // Abre el fichero de configuración para lectura
20   ifstream fc("colores.ini");
21
22   if (!fc) {
23     cerr « "Error al abrir colores.ini" « endl;
24     return 1;
25   }
26
27   string bufer;  // para leer cada línea
28   cout.setf(ios_base::uppercase); // mayúsculas
29   // Bucle de lectura línea a línea
30   while (getline(fc, bufer)) {
31     // Creamos un flujo en memoria sobre bufer
32     istringstream linea(bufer);
33     // Leemos «color» y los componentes de «linea»
34     linea » color » rojo » verde » azul;
35     cout « color « ":\t(" « hex
36          « rojo « "," « verde « ","
37          « azul « ")" « endl;
38   }
39 }
```

7. Empleo de los estados de los flujos

Cada flujo tiene asociado un estado que cambia en cada operación y que puede ser examinado para ver si ésta ha resultado bien o no. El estado de un

flujo se representa como un conjunto de indicadores. Como la mayor parte de las constantes utilizadas para expresar el comportamiento de los flujos, esos indicadores se definen en la clase base *ios_base* de *basic_ios*:

```
class ios_base {
public:
  // ...
  static const iostate badbit,    // el flujo está corrompido
                       eofbit,    // fin de archivo
                       failbit,   // la operación sig. va a fallar
                       goodbit,   // goodbit == 0
  // ...
};
```

El estado de flujo se encuentra en la clase base *basic_ios* de *basic_stream* en `<ios>`:

```
bool good() const;        // la operación siguiente podría tener
                          //  éxito
bool eof() const;         // ha llegado al final de la entrada
bool fail() const;        // la operación siguiente fallará
bool bad() const;         // el flujo está corrompido
iostate rdstate() const;  // devuelve los indicadores de estado
void clear(iostate f = goodbit);
                          // establece indicadores de estado
void setstate(iostate f) { clear (rdstate() | f); }
                          // suma f al estado
bool operator !() const;  // no good()
operator void*() const;   // distinto de cero si good()
```

Los significados de los estados son los siguientes:

good() La operación precedente de E/S funcionó bien, y la siguiente debería de hacerlo.

eof() La operación precedente devolvió una condición de fin de fichero: se acabaron los datos de entrada, no había más que leer.

fail() La operación precedente es inválida. Pero si el estado no es también *bad()*, entonces aún se puede utilizar el flujo si la condición de error es corregida.

bad() La operación precedente falló y el flujo está corrompido. No hay ninguna posibilidad de recuperarlo.

Los dos operadores sirven para comprobar un flujo directamente. El operador *operator* `void *` es en realidad un método de conversión que convierte un flujo en un puntero genérico. Podría estar hecho así:

```
ios_base::operator void*()
{
  return fail() ? 0 : static_cast<void*>(this);
}

// Ejemplo de empleo

if (cin >> c)
```

```
  // ...procesar c; lectura OK
else
 // ...fallo de lectura
```

El operador *operator* `!` hace lo contrario, pero convirtiendo a `bool` para poder utilizarlo en una comprobación. Estaría hecho así:

```
bool ios_base::operator !() { return fail(); }

// Ejemplo de empleo:

cin >> a;
if (!cin) cerr "Error leyendo 'a'" << endl;
```

El siguiente programa ejemplo cuenta el número de «palabras» de la entrada estándar. De nuevo se entiende por «palabra» un conjunto consecutivo de caracteres no blancos; normalmente el programa se empleará redirigiendo la entrada estándar desde un fichero de texto.

cntplbrs.cpp

```
1  // Modo de empleo: cntplbrs < fichero
2
3  #include <iostream>
4  #include <cctype>    // Por isspace()
5  using namespace std;
6
7  int main()
8  {
9    int contador_palabras = 0;
10   int sgute_palabra();
11
12   while (sgute_palabra())
13     ++contador_palabras;
14   cout « "Número de \"palabras\": "
15        « contador_palabras « endl;
16 }
17
18 // Lee caracteres que forman una palabra
19 // Devuelve la longitud de la palabra
20 int sgute_palabra()
21 {
22   char c;
23   int longitud_palabra;
24
25   cin » c;
26   for (longitud_palabra = 0;
27        !cin.eof() && !isspace(c);
28        ++longitud_palabra)
29     cin.get(c);
30   return longitud_palabra;
31 }
```

8. Mezcla de las bibliotecas de C y de C++

A pesar de todo lo visto puede que a un programador en C le resulten más cómodas, por estar ya habituado, algunas funciones de la biblioteca estándar de C. No hay nada malo en ello, ya se han mencionado las ventajas y desventajas de cada una al principio. Pero si se está tentado de mezclar funciones de la biblioteca de C con las de `<iostream>` puede haber problemas de sincronización debido a que usen distintas estrategias de manejo de búferes[3]. Esto puede evitarse con el método *ios_base::sync_with_stdio()* (el nombre es largo pero descriptivo).

Ejemplo:

esmix.cpp

```
1  // Mezclando las bibliotecas de E/S de C y C++
2
3  #include <cstdio>
4  #include <iostream>
5  using namespace std;
6
7  unsigned long int factorial(int n);
8
9  int main()
10 {
11   int n;
12
13    ios_base::sync_with_stdio();
14
15   do {
16     cout « "\nIntroduzca un número positivo n "
17          « "ó 0 para parar: ";
18     scanf("%d", &n);
19     printf("\n%d! = %lu", n, factorial(n));
20   } while (n > 0);
21   cout « "\nFin de la sesión" « endl;
22   return 0;
23 }
24
25 unsigned long int factorial(int n)
26 { // versión iterativa
27   unsigned long int f = 1ul;
28
29   for (int i = 2; i <= n; ++i)
30     f *= i;
31   return f;
32 }
33
```

[3]Tamponamiento, *bufering*.

9. Conclusión

La biblioteca de E/S de C++, llamada *IOStreams*, proporciona una forma segura y fiable de utilizar E/S mediante los *flujos*. Es preferible a la de C, llamada *stdio*[4], por la mejor comprobación de tipos. Para la escritura de clases proporciona métodos *naturales* de impresión mediante la sobrecarga de los operadores de inserción y extracción.

Los programadores con experiencia en C se resisten al uso de *IOStreams* y sólo la utilizan para lo más básico. Vale la pena dedicarle un poco de tiempo y hacer buen uso de ella porque tiene mucho que ofrecer y, cuando uno se acostumbra un poco, su empleo es más natural y claro. Compare por ejemplo la impresión de un número:

```
printf("%d\n", numero);
cout << numero << endl;
```

y compare la impresión de una variable de un TAD; por ejemplo, un complejo:

```
printf("%g%+gi", z.re, z.im);
cout << z;        // una vez sobrecargado <<, claro
```

[4]No se confunda: el fichero de cabecera `<cstdio>` es eso, un fichero de cabecera, no una biblioteca de funciones.