

Metodología de la Programación

Tema 6. Gestión de E/S. Ficheros

Departamento de Ciencias de la Computación e I.A.



DECSAI
Universidad de Granada



ugr

Universidad
de Granada

ETSIIIT Universidad de Granada

Curso 2014-15

Contenido del tema I

Parte I: E/S en C++

- 1 Introducción
 - Flujos de E/S
 - Clases y ficheros de cabecera
 - Flujos y buffers
- 2 Entrada/salida con formato
 - Introducción
 - Banderas de formato
 - Modificación del formato
 - Manipuladores de formato
- 3 Entrada/salida sin formato
 - Salida sin formato
 - Entrada sin formato
 - Devolución de datos al flujo
 - Consultas al flujo
 - Ejemplos de `read()` y `write()`
- 4 Estado de los flujos
 - Banderas de estado
 - Operaciones de consulta y modificación
 - Flujos en expresiones booleanas
- 5 Restricciones en el uso de flujos

Contenido del tema II

Parte II: ficheros

- 6 Flujos asociados a ficheros
 - Introducción
 - Tipos de ficheros
 - Apertura y cierre de ficheros
 - Modos de apertura de ficheros
 - Ejemplos de programas con ficheros binarios y de texto
 - Operaciones de posicionamiento
 - Clase `fstream`

- 7 Flujos asociados a strings

Parte I

E/S en C++

Índice

- 1 Introducción
 - Flujos de E/S
 - Clases y ficheros de cabecera
 - Flujos y buffers

2 Entrada/salida con formato

3 Entrada/salida sin formato

4 Estado de los flujos

5 Restricciones en el uso de flujos

Conceptos básicos

En algunos lenguajes de programación las sentencias necesarias para realizar operaciones de entrada salida forman parte del propio lenguaje. Esto no es así en C++: la entrada/salida se basa en un conjunto de librerías: **iostream** y **fstream**

Conceptos básicos

El tratamiento de la E/S en C++ se basa en el concepto de flujo de bytes:

- los programas recibe bytes de flujos de entrada. Estos bytes pueden provenir de un teclado, de un dispositivo de entrada, de un disco, de otro programa, etc.
- los programas insertan bytes en los flujos de salida. Los bytes podrían ir a la pantalla, a una impresora, un dispositivo de almacenamiento u otro programa.

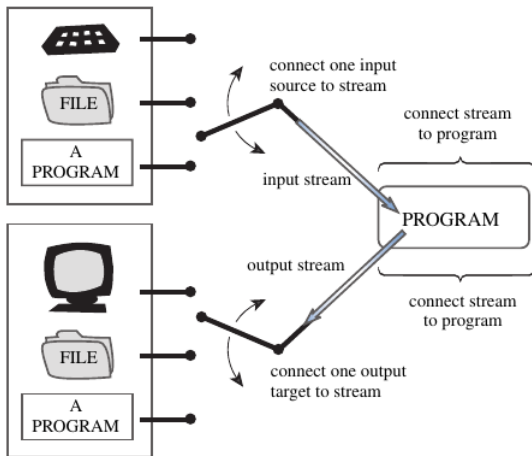
De esta forma un flujo actúa como un intermediario entre el programa y los dispositivos de entrada y salida. El flujo permite tratar estos orígenes o destinos de datos de forma homogénea.

Conceptos básicos

Otro concepto básico es el de **buffer**: bloque de memoria temporal e intermedia que facilita la transmisión de información entre programas y dispositivos. El uso de esta memoria se debe a la diferente velocidad de funcionamiento de programas y dispositivos de entrada/salida.

- supongamos que un programa debe procesar el contenido de un archivo. En lugar de leer byte a byte, se suele leer un bloque completo, que se almacena en esta memoria, y de donde va leyendo el programa. Así se reduce el número de accesos al dispositivo. ¿Cuándo volverá a leerse?
- el teclado proporciona un carácter cada vez, así que no sería necesario, en principio, el uso de **buffer**. Sin embargo, su uso se justifica por la posibilidad de modificar los datos de entrada (borrando, por ejemplo) antes de enviarlos pulsando la tecla **Enter**.

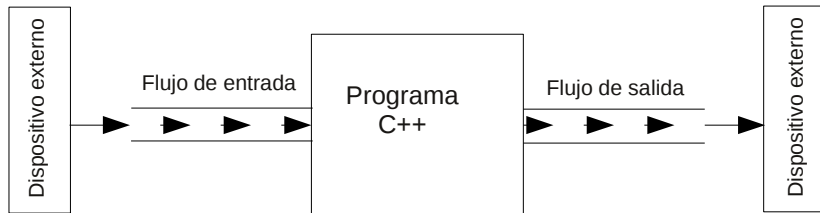
Conceptos básicos



Flujos de E/S

Flujo (stream)

Es una *abstracción* que representa cualquier fuente o destino de datos y que permite realizar operaciones de E/S con él (enviar o recibir datos): podemos verlo como una secuencia de bytes que fluye desde o hacia algún dispositivo.



Flujos de E/S

Consideraciones:

- el flujo oculta los detalles de lo que ocurre con los datos en el dispositivo de E/S real.
- un flujo siempre está asociado a un dispositivo sobre el que actuar.
- es frecuente que se trate de un dispositivo físico (teclado, fichero de disco, pantalla, impresora) aunque podría ser otro programa

Tipos de flujos

Así puede hablarse de:

- **flujos de entrada:** la secuencia de bytes fluye desde un dispositivo de entrada (teclado, fichero de disco, conexión de red, etc) hacia el programa.
- **flujos de salida:** la secuencia de bytes fluye desde el programa hacia un dispositivo de salida (pantalla, fichero de disco, impresora, conexión de red, otro programa, ...).
- **flujos de entrada/salida:** la secuencia de bytes puede fluir en ambos sentidos.

Tamaño finito de los flujos

Puede considerarse que un flujo de datos es una secuencia de n caracteres consecutivos.

- una vez leídos los n caracteres se da por finalizada la operación y la lectura de un nuevo carácter produciría un error.
- por ejemplo, si usamos `get()`, se devolverá la constante especial EOF (*end of file*) cuando quiere seguir leyendo tras haber leído todos sus caracteres.
- así, podemos ver el flujo como una secuencia de n caracteres, seguida por la constante EOF.



Flujos estándar

El uso de los flujos precisa la inclusión del archivo de cabecera `<iostream>`. Además, cada flujo tiene asociado un dispositivo sobre el que actúa. Los flujos estándar predefinidos son:

- **cin**: instancia de `istream` conectado a la entrada estándar (teclado).
- **cout**: instancia de `ostream` conectado a la salida estándar (pantalla).
- **cerr**: instancia de `ostream` conectado a la salida estándar de error sin buffer (pantalla).
- **clog**: instancia de `ostream` conectado a la salida estándar de error (pantalla).

Redireccionamiento de entrada/salida

Los flujos estándar de entrada y salida usualmente conectan con el teclado y con la pantalla respectivamente. Pero los sistemas operativos proporcionan mecanismos para cambiar los dispositivos con los que conectan los flujos de entrada, salida y error.

Redirecciones

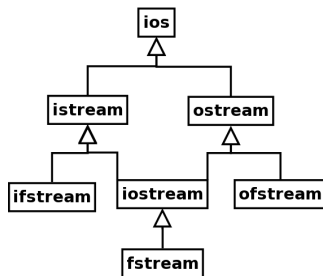
- salida: programa > salida.txt. Ejemplo: `ls -al > contenido`
- entrada: programa < entrada.txt
- salida de error: programa >& error.txt. Ejemplo: `g++ -o error error.cpp >& errores`
- todo: (programa < entrada.txt > salida.txt) >& fichero.txt
- encauzamiento: programa1 < entrada1.txt | programa2 | programa3 > salida3.txt. Ejemplo: `ls -al | wc -w`

Ficheros de cabecera

Todo lo que se define en estos ficheros está incluido en el espacio de nombres (*namespace*) `std`.

- `<istream>`: definición de la clase `istream` (flujos de entrada)
- `<ostream>`: definición de la clase `ostream` (flujos de salida)
- `<iostream>`: declara los servicios básicos requeridos en todas las operaciones de E/S con flujos:
 - incluye a `<istream>` y `<ostream>`.
 - definición de la clase `iostream` (gestión de flujos de E/S).
 - declaración (y creación) de los flujos estándar: `cin` (`istream`) y `cout`, `cerr`, `clog` (`ostream`)
- `<iomanip>`: declara servicios usados para la E/S con formato (manipuladores tales como `setw()` y `setprecision()`).
- `<fstream>`: E/S con ficheros.

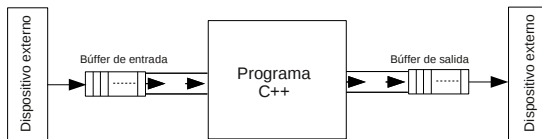
Jerarquía de clases



- **ios**: Superclase abstracta.
- **istream**: flujo de entrada (**cin** es de esta clase).
- **ostream**: flujo de salida (**cout**, **cerr** y **clog** son de esta clase).
- **iostream**: flujo de entrada y salida.
- **ifstream**: flujo de entrada desde fichero.
- **ofstream**: flujo de salida hacia fichero.
- **fstream**: flujo de E/S con ficheros.

Flujos y buffers

- las operaciones de E/S con dispositivos suelen ser lentas en comparación con la CPU o la transferencia a memoria
- para aumentar la eficiencia, los dispositivos de E/S no se comunican directamente con nuestro programa, sino que usan un **buffer intermedio** para almacenamiento temporal de los datos
- cuando el buffer se llena, se hace la transferencia a o desde el dispositivo
- el método **ostream::flush()** o bien **endl** ordenan la transferencia inmediata de un buffer de salida al dispositivo



Flujos y buffers I

- Al hacer cout de una cadena, no aparecerá en pantalla hasta que se llene el buffer o usemos `ostream::flush()` o `endl`.

```
#include<iostream>
```

```
using namespace std;
```

```
int main(){
```

```
    // Se muestra un mensaje por pantalla, pero no se vera
```

```
    // porque no se llena el buffer
```

```
    cout << "Prueba";
```

```
    // Ahora aseguramos que el programa entra en bucle
```

```
    // infinito, para dar tiempo a que pudiera salir el
```

```
    // mensaje por pantalla
```

```
    while(true);
```

```
}
```

Flujos y buffers I

Haciendo uso de **flush** o forzando la salida de **endl** se soluciona este problema (en ambos casos se fuerza el vaciado del buffer):

```
#include<iostream>
using namespace std;
int main(){
    // Se muestra un mensaje por pantalla, pero no se vera
    // porque no se llena el buffer
    cout << "Prueba";

    // Forzamos su salida del buffer mediante flush
    cout.flush();

    // Ahora aseguramos que el programa entra en bucle
    // infinito, para dar tiempo a que pudiera salir el
    // mensaje por pantalla
    while(true);
}
```

Flujos y buffers

Sin embargo, **cerr** es un flujo que no espera a que se llene su buffer para transferir los datos (se mostrará por pantalla el mensaje según se envía al buffer).

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

int main(){
    // Este mensaje se mostrara directamente, al volcarse
    // el flujo de error de forma inmediata
    cerr << "Prueba";

    // Se fuerza el bucle infinito para ver qué ocurre con
    // el mensaje
    while(true);
}
```

Índice

1 Introducción

2 Entrada/salida con formato

- Introducción
- Banderas de formato
- Modificación del formato
- Manipuladores de formato

3 Entrada/salida sin formato

4 Estado de los flujos

5 Restricciones en el uso de flujos

Introducción a entrada/salida con formato

E/S con formato

- implica transformar los caracteres (bytes) que se leen o escriben al tipo de dato deseado
- se pasa de la representación interna de los datos a una representación comprensible por los usuarios (una secuencia de caracteres imprimibles). Por ejemplo al mostrar un dato `double` en pantalla con `cout` se transforma la representación interna del dato en un conjunto de caracteres imprimibles
- se hace a través de los operadores de inserción y extracción de flujos: `>>` y `<<`

E/S sin formato

- la información se transfiere en bruto, sin transformaciones.

Introducción a entrada/salida con formato I

Veamos un ejemplo del uso del formato:

```
#include <iostream>
using namespace std;
int main(){
    // Cadena
    const char* S1 = "AEIO";

    // Valor de tipo flotante
    float S2 = 12.4;

    // Se escribe S1 sin formato, indicando que se insertan
    // 4 caracteres en el flujo. Aqui no habra diferencia,
    // ya que los caracteres son imprimibles
    cout << "Salida sin formato....." << endl;
    cout.write(S1, 4);
    cout << endl;
```


Introducción a entrada/salida con formato II

```
// Se muestra el contenido de S2, indicando la direccion  
// de la que extraer los caracteres. El numero de caracteres  
// a incluir es el dado por float  
cout.write(reinterpret_cast<const char*>(&S2),  
           sizeof(float));  
  
// Forzamos la salida mediante endl  
cout << endl;  
  
// Ahora se hace la salida con formato  
cout << "Salida con formato....." << endl;  
cout << S1 << endl;  
cout << S2 << endl;  
}
```

Introducción a entrada/salida con formato I

La salida obtenida es:

```
Salida sin formato.....  
AEIO  
ffFA  
Salida con formato.....  
AEIO  
12.4
```

El operador <<

Ya lo hemos usado, pero conviene considerar:

- se conoce como el **operador de inserción en flujos**
- cuando el compilador de C++ encuentra una expresión como la siguiente, busca la versión del método a llamar en función del tipo de la variable **dato**.

```
cout << dato;
```

- esta sentencia hace que el valor de la variable **dato** se envíe desde memoria hacia el flujo de salida (salida estándar en este caso) transformado en caracteres imprimibles.

El operador <<

- el operador devuelve una referencia al mismo objeto ostream usado. Esto permite encadenar salidas.
- el operador está sobrecargado para los tipos fundamentales de C++ (cadenas tipo C, strings y punteros)
- además, como hemos visto en el tema anterior, lo podemos sobrecargar para nuestros propios tipos.

Consideraremos un ejemplo de salida con diferentes tipos de datos.

Ejemplo operador << |

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string cads="Hola";
    int x=74;
    double y=65.1234;
    int *ptr=&x;
    bool valor=false;
    const char *adios = "Adios";
    char cadena[100]="Cadena";

    cout << "Cadena tipo string, cads: " << cads << endl;
    cout << "Valor de tipo entero, x:" << x << endl;
    cout << "Valor de tipo double, y: " << y << endl;
    cout << "Puntero a entero, ptr (direccion): " << ptr << endl;
    cout << "Tipo bool asignado a false, valor: " << valor << endl;
```

Ejemplo operador << II

```
// Al recibir char* lo interpreta como la cadena asociada....
cout << "Cadena de caracteres, adios (sin cast): " << adios << endl;
// Si queremos ver la direccion hay que hacer cast ....
cout << "Cadena de caracteres, adios (con cast): " <<
    static_cast<const void *>(adios) << endl;
cout << "Array de caracteres, cadena: " << cadena << endl;
}
```

Ejemplo operador << |

La salida obtenida es:

```
Cadena tipo string, cads: Hola
Valor de tipo entero, x:74
Valor de tipo double, y: 65.1234
Puntero a entero, ptr (direccion): 0x7ffff91b7bec
Tipo bool asignado a false, valor: 0
Cadena de caracteres, adios (sin cast): Adios
Cadena de caracteres, adios (con cast): 0x40108d
Array de caracteres, cadena: Cadena
```

El operador >>

A recordar:

- se conoce como el **operador de extracción de flujos**
- cuando el compilador de C++ encuentra la expresión siguiente, busca la versión a llamar dependiendo del tipo de la variable **dato**.

```
cin >> dato;
```

- la sentencia lee un dato del flujo de entrada (entrada estándar en este caso), lo transforma al tipo de la variable **dato** y lo almacena en ella

El operador >>

- el operador devuelve una referencia al mismo objeto `istream` usado: esto permite encadenar entradas.
- el operador está sobrecargado para los tipos fundamentales de C++, cadenas tipo C, strings y punteros
- además, puede sobrecargarse para nuestros propios tipos

operator >>: Ejemplo

- El operador >> está implementado de forma que elimina los caracteres separadores (blanco, tab y enter) que haya en el flujo antes del dato.
- Cuando se usa para leer una cadena, lee hasta que encuentra un separador.

operator >>: Ejemplo I

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    char cad[100];
    int x;
    float y;

    // Se leen datos que se vuelcan en un entero y un float
    cin >> x >> y;
    // Se leen datos que se vuelcan en un cadena de caracteres
    cin >> cad;

    // Se muestra el resultado de la lectura
    cout << "Entero, x: " << x << endl;
    cout << "Float, y: " << y << endl;
    cout << "Cadena de caracteres, cad: " << cad << endl;
}
```

operator >>: Ejemplo I

Ante la entrada:

123

456.789

Hola Pepe

```
Entero, x: 123
```

```
Float, y: 456.789
```

```
Cadena de caracteres, cad: Hola
```

operator >>: Ejemplo I

Ante la entrada:

123.234

22

Hola Pepe

```
Entero, x: 123
```

```
Float, y: 0.234
```

```
Cadena de caracteres, cad: 22
```

Banderas de formato I

Cada flujo tiene asociadas una serie de banderas (indicadores) de formato para controlar la apariencia de los datos que se escriben o la forma de procesar los que se leen.

Son enumerados (**enum**) de tipo **fmtflags** (`long int`) definidas en la clase `ios_base` (superclase de `ios`). Se incluye a continuación una tabla con las banderas disponibles:

```
enum fmtflags {
    boolalpha=1L<<0,      dec=1L<<1,      fixed=1L<<2,      hex=1L<<3,
    internal =1L<<4,      left =1L<<5,      oct=1L<<6,        right=1L<<7,
    scientific=1L<<8,     showbase =1L<<9,   showpoint=1L<<10, showpos =1L<<11,
    skipws  =1L<<12,     unitbuf  =1L<<13,   uppercase=1L<<14,
    adjustfield= left | right | internal,
    basefield= dec | oct | hex,
    floatfield = scientific | fixed,
};
```

Utilizando el operador **OR** a nivel de bits (`|`) se pueden definir varias banderas en una sola variable de tipo `fmtflags`: `oct|left|showbase`.

Banderas de formato I

left	Salida alineada a la izquierda
right	Salida alineada a la derecha
internal	Se alinea el signo y los caracteres indicativos de la base por la izquierda y las cifras por la derecha
dec	Entrada/salida decimal para enteros (valor por defecto)
oct	Entrada/salida octal para enteros
hex	Entrada/salida hexadecimal para enteros
scientific	Notación científica para coma flotante
fixed	Notación normal (punto fijo) para coma flotante
skipws	Descartar blancos iniciales en la entrada
showbase	Se muestra la base de los valores numéricos: 0 (oct), 0x (hex)
showpoint	Se muestra el punto decimal
uppercase	Los caracteres de formato aparecen en mayúsculas
showpos	Se muestra el signo (+) en los valores positivos
unitbuf	Salida sin buffer (se vuelca con cada operación)
boolalpha	Leer/escribir valores bool como strings alfabéticos (true y false)

Modificación del formato

La clase `ios_base` dispone de métodos para modificar o consultar las banderas de formato y devolver el estado de las mismas antes del cambio. Se tratan de métodos generales que pueden aplicarse a cualesquiera de los modificadores de formato vistos en la tabla anterior.

- `fmtflags setf(fmtflags banderas):`
 - `banderas` es un conjunto de una o más banderas unidas con el operador OR lógico a nivel de bits.
- `fmtflags setf(fmtflags banderas, fmtflags mascara):`
 - se usa para activar una de las banderas de un grupo, usando una máscara en el parámetro `mascara`.

banderas	máscara
left, right o internal	adjustfield
dec, oct o hex	basefield
scientific o fixed	floatfield

- Devuelve el estado de las banderas anterior al cambio.

Modificación del formato I

Veremos varios ejemplos de uso. Comenzamos por ejemplo fijando el formato científico o fijo (normal). Estas especificaciones se realizan sobre la máscara llamada **floatfield**.

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

int main(int argc, char *argv[]) {
    // Se muestra valor de forma usual
    cout << "Salida en forma usual: " << 123.45 << endl;

    // Se fija formato científico
    cout.setf(ios::scientific, ios::floatfield);
    cout << "salida en formato científico: " << 123.45 << endl;

    // Se revierte a formato punto fijo, donde el numero
    // de cifras decimales a mostrar es 6 por defecto
    cout.setf(ios::fixed, ios::floatfield);
    cout << "Salida en punto fijo, 6 decimales: " << 123.45 << endl;
}
```

Modificación del formato I

La salida producida es:

```
Salida en forma usual: 123.45  
salida en formato cientifico: 1.234500e+02  
Salida en punto fijo, 6 decimales: 123.450000
```

Modificación del formato I

También podemos forzar la aparición del signo, mediante **showpos**:

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

int main(int argc, char *argv[]) {
    // Se muestra de forma habitual
    cout << "Salida en forma usual: " << 123 << endl;

    // Se fuerza la salida del signo
    cout.setf(ios::showpos);
    cout << "Salida forzando presencia de signo: " << 123 << endl;
}
```

Modificación del formato I

La salida producida es:

```
Salida en forma usual: 123  
Salida forzando presencia de signo: +123
```

Modificación del formato I

Podemos mostrar un valor con bases diferentes:

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

int main(int argc, char *argv[]) {
    // Se visualiza de forma normal
    cout << "Valor en modo normal: " << 123 << endl;

    // Se fija la visualizacion en hexadecimal
    cout.setf(ios::hex,ios::basefield);
    cout << "Valor en hexadecimal: " << 123 << endl;

    // Seguimos en hexadecimal, pero al ver los numeros
    // se mostrara tambien la base: 0x para hexadecimal
    // y 0 para octal
    cout.setf(ios::showbase);
    cout << "Seguimos hexadecimal + se muestra la base: " << 123 << endl;
```

Modificación del formato II

```
// Se cambia a octal
cout.setf(ios::oct,ios::basefield);
cout << "Valor en octal + se muestra la base: " << 123 << endl;

// Se resetean para que no se vea la base, pero seguimos en octal
cout.setf(ios::fmtflags(0),ios::showbase);
cout << "Valor en octal, sin base: " << 123 << endl;
}
```

Modificación del formato I

La salida producida es:

```
Valor en modo normal: 123
Valor en hexadecimal: 7b
Seguimos hexadecimal + se muestra la base: 0x7b
Valor en octal + se muestra la base: 0173
Valor en octal, sin base: 173
```

Modificación del formato

Otras funciones disponibles:

- `void unsetf(fmtflags banderas)`
 - permite desactivar banderas
 - banderas: conjunto de una o más banderas (agregadas mediante el operador `|`).
- `fmtflags flags()` `const`
 - devuelve las banderas actualmente activas
- `fmtflags flags(fmtflags banderas)` `const`
 - establece nuevas banderas en el flujo, borrando todas las que hubiera con anterioridad
 - devuelve las banderas anteriores al cambio

Modificación del formato I

Si deseamos resetear el comportamiento anterior y mostrar valores ajustados a la derecha, en formato hexadecimal y mostrando la base, haríamos:

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

int main () {
    // Se resetean los flags anteriores para ajustar a la derecha,
    // mostrar en hexadecimal y que se vea la base
    ios::fmtflags previos=cout.flags ( ios::right | ios::hex | ios::showbase );

    // Se fija ademas un ancho de 10 caracteres para la salida
    cout.width (10);

    // Se muestra el valor 10 con estas características
    cout << 100 << endl;

    // Se muestran los flag previos
    cout << "Flags previos: " << previos << endl;
```

Modificación del formato II

```
// Se muestran los actuales  
ios::fmtflags actuales=cout.flags();  
cout << "Flags actuales: " << actuales << endl;  
return 0;  
}
```

El resultado de estas operaciones es:

```
0x64  
Flags previos: 0x1002  
Flags actuales: 0x288
```

Modificación del formato

También hay métodos específicos que permiten modificar ciertos aspectos del formato (todos ellos devuelven el valor previo a la modificación):

- `precision()`
 - `streamsize precision() const`: devuelve la precisión actual (máximo número de dígitos al escribir números reales).
 - `streamsize precision(streamsize prec)`: establece la precisión
- `fill()`
 - `char fill() const`: devuelve el carácter de relleno usado al justificar a izquierda o derecha (espacio en blanco por defecto).
 - `char fill(char fillch)`: establece el carácter de relleno.
- `width()`
 - `streamsize width() const`: devuelve el valor de la anchura de campo (mínimo número de caracteres a escribir).
 - `streamsize width(streamsize anchura)`: establece la anchura de campo. Sólo afecta a la siguiente operación de salida.

Modificación del formato I

Se muestra un ejemplo en el que se manipula el ancho usado para la salida:

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

int main() {
    // Cambio el ancho a 20
    cout.width(20);

    // Se fija el caracter de relleno: un punto
    cout.fill('.');

    // Se indica que se justifica a la derecha
    cout.setf(ios::right, ios::adjustfield);
    cout << 123.45 << endl;

    // Ahora se muestra de forma normal: los 20 caracteres eran
    // para la salida anterior
    cout << 123.45 << endl;
```

Modificación del formato II

```
// Fijamos ancho a 10 caracteres  
cout.width(10);  
cout << 123.45 << endl;  
}
```

Resultado:

```
.....123.45  
123.45  
....123.45
```

Manipuladores de formato

Consideraciones sobre los manipuladores de formato:

- son constantes y métodos que permiten modificar también las banderas de formato
- se usan en la propia sentencia de entrada o salida (<< o >>).
- para usar los que no tienen argumentos, incluiremos <iostream>
- Para usar los que tienen argumentos, incluiremos <iomanip>
- `setw(int n)` afecta sólo a la siguiente operación de salida.

Manipuladores de formato

Banderas básicas de formato

boolalpha	noboolalpha
showbase	noshowbase
showpoint	noshowpoint
showpos	noshowpos
skipws	noskipws
unitbuf	nounitbuf
uppercase	nouppercase

Banderas de base numérica

dec	hex	oct
-----	-----	-----

Banderas punto flotante

fixed	scientific
-------	------------

Banderas ajuste de formato

internal	left	right
----------	------	-------

Extracción de blancos

WC

Manipuladores de salida

endl	ends	flush
------	------	-------

Manipuladores con parámetros

```
setiosflags(fmtflags mask)
resetiosflags(fmtflags mask)
setbase(int n)
setfill(int n)
setprecision(int n)
setw(int n)
```

Manipuladores de formato I

Veamos un ejemplo de uso:

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

int main(int argc, char *argv[]) {
    // Se fija base hexadecimal y se indica que debe verse la
    // base. Antes modificados con setf
    cout << "Base hexadecimal y visualizacion de la base, valor 20 " << endl;
    cout << setbase(16) << showbase << 20 << endl;

    // Tambien con manipulador hex sin argumentos
    cout << "\nRealizada la misma operacion con manipulador sin arg. " << endl;
    cout << hex << 20 << endl;

    // Se indica base octal y que no se vea la base
    cout << "\nBase octal y sin visualizacion de la base, valor 20 " << endl;
    cout << oct << noshowbase << 20 << endl;
```


Manipuladores de formato II

```
// Se muestra en decimal el valor hexadecimal 0x20
cout << "\nValor en decimal del valor hexadecimal 0x20 " << endl;
cout << dec << 0x20 << endl;

// Se fija la precision a valor 2
cout << "\nVisualizando valor 2.123456 con precision igual a 3 " << endl;
cout << setprecision(3) << 2.123456 << endl;

// Se fija el ancho a 20 y se muestra el mismo valor
cout << "\nFijando ancho a 10 y visualizacion de 2.123456 " << endl;
cout << setw(10) << 2.123456 << endl;

// Igual que la sentencia anterior, pero justificando a la izquierda
cout << "\nIgual que valor previo, con justif. a izquierda " << endl;
cout << setw(10) << left << 2.123456 << endl;

// Se agrega completar espacios sobrantes con *
cout << "\nSe agregan * para rellenar espacio sobrante " << endl;
cout << setw(10) << left << setfill('*') << 2.123456 << endl;
```

Manipuladores de formato III

```
// Ahora con justificacion a la derecha  
cout << "\nIgual pero con justificacion a derecha " << endl;  
cout << setw(10) << right << setfill('*') << 2.123456 << endl;  
}
```

Resultado:

```
Base hexadecimal y visualizacion de la base, valor 20  
0x14  
  
Realizada la misma operacion con manipulador sin arg.  
0x14  
  
Base octal y sin visualizacion de la base, valor 20  
24  
  
Valor en decimal del valor hexadecimal 0x20  
32  
  
Visualizando valor 2.123456 con precision igual a 3  
2.12
```

Manipuladores de formato IV

```
Fijando ancho a 10 y visualizacion de 2.123456  
2.12
```

```
Igual que valor previo, con justif. a izquierda  
2.12
```

```
Se agregan * para rellenar espacio sobrante  
2.12*****
```

```
Igual pero con justificacion a derecha  
*****2.12
```

Índice

- 1 Introducción
- 2 Entrada/salida con formato
- 3 Entrada/salida sin formato
 - Salida sin formato
 - Entrada sin formato
 - Devolución de datos al flujo
 - Consultas al flujo
 - Ejemplos de `read()` y `write()`
- 4 Estado de los flujos
- 5 Restricciones en el uso de flujos

ostream::put()

Se trata de realizar la entrada/salida como manipulación de bytes.

Método `ostream& put(char c);`

- envía el carácter `c` al objeto `ostream`
- devuelve una referencia al flujo usado

Salida sin formato: **put** |

Ejemplo de uso:

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

int main(int argc, char *argv[]) {
    char c1='a', c2='b';

    // Se envia c1 a cout
    cout.put(c1);

    // Se envia c2 a cout
    cout.put(c2);

    // Se envia 'c' a cout
    cout.put('c');

    // Se envia salto de linea a cout
    cout.put('\n');
```

Salida sin formato: **put** II

```
// Se muestra todo seguido  
cout.put(c1).put(c2).put('c').put('\n');  
  
// Se muestra de la forma habitual  
cout << c1 << c2 << 'c' << '\n';  
}
```

Resultado:



```
abc  
abc  
abc
```

Salida sin formato: `ostream::write()`

Método `ostream& write(const char* s, streamsize n);`

- envía al objeto `ostream` el bloque de datos apuntados por `s`, con un determinado número de caracteres (**n**).
- devuelve una referencia al flujo
- suele usarse con flujos asociados a ficheros y no con `cout`

Salida sin formato: **write** |

Ejemplo de uso:

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

int main(int argc, char *argv[]) {
    const char *cad="Hola";
    int x=0x414243;

    // Se muestran 4 caracteres de la cadena
    cout.write(cad,4);

    // Se muestran los 6 caracteres, incluyendo el salto de linea
    cout.write("Adios\n",6);

    // Se muestra el valor almacenado en x. Hay que hacer conversion
    // a char * de la direccion de x, de forma que se mostrarian los
    // 4 primeros bytes (puede que alguno no sea imprimible....)
    cout.write((char*)&x,4);
```

Salida sin formato: **write** II

```
// Se salta de linea  
cout << endl;  
  
// Se pasa a considerar el siguiente byte y a partir de ahí se  
// muestran 2  
cout.write(((char*)&x)+1,2);  
}
```

Resultado:

```
HolaAdios  
CBA  
BA
```

Entrada sin formato: `istream::get()`

Método `int get()`;

- extrae un carácter del flujo y devuelve su valor convertido a entero
- devuelve EOF (End Of File) si leemos más allá del último carácter disponible en el flujo
- EOF es una constante definida en `<iostream>` que suele tener el valor -1. Está asociada a la combinación de teclas Ctrl+D en linux, y Ctrl+Z en Windows.

Entrada sin formato: `get` I

Ejemplo de uso:

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

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

    // Se tecleara qwerty
    // Se lee un caracter
    ci = cin.get();

    // Se muestra lo leído
    cout << "Valor leído: " << ci << " Pasado a caracter: " << (char)ci << endl;

    // Se lee otro caracter y se almacena en un caracter
    cc = cin.get();

    // Se muestra lo leído
```

Entrada sin formato: `get` II

```
    cout << "Valor leído: " << cc << " Pasado a entero: " << (int)cc << endl;  
}
```

Resultado:

```
Valor leído: 113 Pasado a caracter: q  
Valor leído: w Pasado a entero: 119
```

Entrada sin formato: `istream::get()`

Método `istream& get(char& c);`

- extrae un carácter del flujo y lo almacena en `c`
- devuelve una referencia al objeto `istream` usado

Entrada sin formato: `get`

Ejemplo de uso:

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

int main(int argc, char *argv[]) {
    char c1,c2,c3;

    // Se teclea qwerty
    // Se leen de forma consecutiva 3 caracteres
    cin.get(c1).get(c2).get(c3);

    // Se muestra lo leído
    cout << c1 << c2 << c3 << endl;
}
```

Resultado:



qwe

Entrada sin formato: `istream::get()`

```
istream& get(char* s, streamsize n, char delim='\n');
```

- extrae caracteres del flujo y los almacena como un c-string en el array `s` hasta que (condiciones enlazadas con OR):
 - hayamos leído $n - 1$ caracteres
 - hayamos encontrado el carácter `delim`
 - hayamos llegado al final del flujo o encontrado algún error de lectura
- el carácter `delim` no es extraído del flujo.
- se añade un carácter `'\0'` al final de `s`
- `s` ha de tener espacio suficiente para almacenar los caracteres
- devuelve una referencia al objeto `istream` usado.

Entrada sin formato: `get` I

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

int main() {
    char c1[50], c2[50], c3[50];

    // Se leen hasta 50 caracteres, que se almacenaran en c1
    cin.get(c1,50);

    // Se leen hasta 50 caracteres: se para al leer a
    cin.get(c2,50,'a');

    // Se leen 50 caracteres: se para al leer t
    cin.get(c3,50,'t');

    // Se muestra el resultado de las tres operaciones de lectura
    cout << "Contenido de c1: " << c1 << endl;
    cout << "Contenido de c2: " << c2 << endl;
    cout << "Contenido de c3: " << c3 << endl;
}
```

Entrada sin formato: `get ll`

Resultado: ejemplo de uso (tecleando dos veces la frase “Hola, como estas?” + ENTER). La primera lectura es completa y `c1` contiene la frase entera. La segunda vez que se teclea se alimenta el buffer de entrada con todos los caracteres: la primera lectura consume hasta el carácter ‘a’ (**Hol**) y la segunda lectura hasta ‘t’ (**a, como es**):

```
Contenido de c1: Hola, como estas?  
Contenido de c2:  
Hol  
Contenido de c3: a, como es
```

Entrada sin formato: `istream::getline()`

```
istream& getline(char* s, streamsize n, char delim='\n');
```

- idéntico a `get()` salvo que `getline()` extrae `delim` del flujo, aunque tampoco lo almacena
- otra diferencia es que activa el bit `failbit` si se alcanza el tamaño máximo sin leer `delim`
- extrae caracteres del flujo y los almacena como un c-string en el array `s` hasta que (condiciones enlazadas con OR):
 - se hayan leído $n - 1$ caracteres
 - se encuentre el carácter `delim`
 - se alcance el final del flujo o se produzca algún error de lectura
- se añade un carácter `'\0'` al final de `s`
- `s` ha de tener espacio suficiente para almacenar los caracteres
- devuelve una referencia al objeto `istream` usado

Entrada sin formato: `getline` I

```
#include <iostream>
using namespace std;
int main()
{
    char c1[50], c2[50], c3[50];

    // Se lee hasta encontrar delimitador o hasta 50 caracteres
    cin.getline(c1,50);

    // Igual, tambien se detiene si aparece a
    cin.getline(c2,50,'a');

    // Igual, tambien se detiene si aparece a
    cin.getline(c3,50,'t');

    // Se muestra lo leído
    cout << "Contenido de c1: " << c1 << endl;
    cout << "Contenido de c2: " << c2 << endl;
    cout << "Contenido de c3: " << c3 << endl;
}
```

Entrada sin formato: `getline` II

Resultado: ejemplo de uso (tecleando dos veces la frase "Hola, como estas?" + ENTER). La primera lectura es completa y `c1` contiene la frase entera (sin el ENTER delimitador, que no se almacena). La segunda vez que se teclaea se alimenta el buffer de entrada con todos los caracteres: la primera lectura consume hasta el carácter 'a' (**Hol**, habiéndose sacado del buffer la letra a) y la segunda lectura hasta 't'(**, como es**):

```
Contenido de c1: Hola, como estas?  
Contenido de c2: Hol  
Contenido de c3: , como es
```

Entrada sin formato: `getline()`

```
istream& getline(istream& is, string& str, char delim='\n');
```

- extrae caracteres del flujo y los almacena como un string
- lee hasta encontrar el delimitador
- el delimitador se extrae pero no se almacena en el string

Entrada sin formato: `getline` I

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

int main(){
    // Cadenas de almacenamiento
    string cad1,cad2;

    // Se hace la primera lectura
    getline(cin,cad1);
    cout << cad1 << endl;

    // Se hace la segunda lectura, hasta m
    getline(cin,cad2,'m');
    cout << cad2 << endl;
}
```

Entrada sin formato: **getline** II

Resultado: ejemplo de uso (tecleando dos veces la frase “Hola, como estas?” + ENTER). La primera lectura es completa y **cad1** contiene la frase entera (sin el ENTER delimitador, que no se almacena). La segunda vez que se teclaea se alimenta el buffer de entrada con todos los caracteres: la lectura consume hasta el carácter ‘m ’ (**Hola, co**, habiéndose sacado del buffer la letra **m**):

```
Hola, como estas?  
Hola, co
```


Entrada sin formato: `istream::ignore()` |

```
istream& ignore(streamsize n=1, int delim=EOF);
```

- extrae caracteres del flujo y no los almacena en ningún sitio
 - hasta que se lean n caracteres.
 - o hasta encontrar el carácter `delim`. En este caso `delim` también es extraído
- devuelve una referencia al objeto `istream` usado

Entrada sin formato: **ignore** |

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string c;
    cin.ignore(6);
    getline(cin,c);
    cout << c << endl;
}
```

Resultado: ejemplo de uso (tecleando “123456789”+ ENTER). Se ignoran los 6 primeros caracteres y luego se hace una lectura del resto de los caracteres mediante **getline**). Al mostrarlos aparece **789** como salida:

A screenshot of a terminal window with a light blue background and a black border. The text "789" is displayed in the terminal, representing the output of the program after ignoring the first six characters of the input "123456789".

istream::read() y istream::readsome() I

```
istream& read(char* s, streamsize n);
```

- extrae un bloque de n caracteres del flujo y lo almacena en el array apuntado por s
- si antes aparece EOF, se guardan en el array los caracteres leídos y se activan los bits `failbit` y `eofbit`
- devuelve una referencia al objeto `istream`
- s debe tener reservada suficiente memoria
- no se añade un carácter `'\0'` al final de s

```
streamsize readsome(char* s, streamsize n);
```

- igual función que `read()`, pero devuelve el número de caracteres extraídos con éxito.

Devolución de datos al flujo I

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

- devuelve el carácter `c` al flujo de entrada
- `c` será por tanto el siguiente carácter a leer
- devuelve una referencia al objeto `istream` usado

```
istream& unget();
```

- devuelve al flujo de entrada el último carácter leído
 - devuelve una referencia al objeto `istream` usado
-
- el número de caracteres consecutivos que pueden ser devueltos al flujo **depende del compilador**.
 - el estándar sólo garantiza uno

Devolución de datos al flujo I

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

int main() {
    unsigned char c;

    // Se extrae el primer caracter para analizar si es un
    // numero o una cadena. Se devuelve al flujo para que
    // la siguiente lectura sea completa
    c = cin.get();
    cin.unget();

    // Control de tipo de entrada
    if (isdigit(c)) {
        int n;
        cout << "Es un numero" << endl;
        cin >> n;
    }
    else {
        string cad;
```

Devolución de datos al flujo II

```
    cout << "Es una cadena" << endl;  
    cin >> cad;  
}  
}
```

Resultado: ejemplo de uso (tecleando "1234" + ENTER). Aparece como salida **1234**, con la indicación correspondiente a número:

```
Es un numero  
1234
```

Si se teclea "Hola, como estas?" + ENTER, se leen caracteres hasta encontrar el espacio en blanco (delimitador), con lo que la salida obtenida es:

```
Es una cadena  
Hola,
```

Consultas al flujo

```
int peek();
```

- lee y devuelve el siguiente carácter del flujo, sin extraerlo
- devuelve EOF si se lee más allá del último carácter del flujo

```
streamsize gcount() const;
```

- devuelve el número de caracteres leídos en la última operación de lectura sin formato (get, getline, ignore, peek, read, readsome, putback y unget)

Consultas al flujo I

```
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char *argv[]) {
    // Se controla si es un dígito: la lectura se hace volcando
    // el resultado de lectura en la variable adecuada (el caracter
    // devuelto por peek no se extrae del flujo)
    if (isdigit(cin.peek())) {
        int n;
        cout << "Es un numero" << endl;
        cin >> n;
    }
    else {
        string cad;
        cout << "Es una cadena" << endl;
        cin >> cad;
    }
}
```


Consultas al flujo II

Resultado: ejemplo de uso (tecleando "1234" + ENTER). Aparece como salida **1234**, con la indicación correspondiente a número:

```
Es un numero  
1234
```

Si se tecléa "Hola, como estas?" + ENTER, se leen caracteres hasta encontrar el espacio en blanco (delimitador), con lo que la salida obtenida es:

```
Es una cadena  
Hola,
```

Conteo de caracteres leídos I

```
#include <iostream>
using namespace std;
int main () {
    const int TAM_BUFFER = 10;
    char buffer[TAM_BUFFER];
    int tam = 0;
    // Bucle de realizacion de lecturas: cada iteracion se
    // leen 10 caracteres (dado por TAM_BUFFER)
    while (cin.read(buffer, TAM_BUFFER)){
        cout << "Leido: " << buffer << endl;
        tam += TAM_BUFFER;
        cout << "Valor de tam: " << tam << endl;
    }
    // Se obtiene el valor devuelto por gcount
    cout << "Valor de tam al salir del bucle: " << tam << endl;
    int tamFinal = cin.gcount();
    cout << "Caracteres leidos al final = " << tamFinal << endl;
    tam+=tamFinal;
    cout << "Caracteres leidos en total = " << tam << endl;
}
```

Conteo de caracteres leídos II

Resultado: ejemplo de uso (tecleando "Esto es una frase corta"+ ENTER) (24 caracteres). Hay dos vueltas de lectura de 10 caracteres y el bucle queda a la espera de recibir EOF (**Ctrl+D** en linux) o a tener caracteres suficientes para leer 10 caracteres. Al pulsar **Ctrl+D** se leen los caracteres restantes y se muestra el resultado (4 caracteres considerando **ENTER**):

```
Leido: Esto es un@
Valor de tam: 10
Leido: a frase co@
Valor de tam: 20
Valor de tam al salir del bucle: 20
Caracteres leídos al final = 4
Caracteres leídos en total = 24
```

Copia de entrada en salida con **read** y **write** I

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

int main () {
    const int TAM_BUFFER = 10;
    char buffer[TAM_BUFFER];

    // Bucle de lectura
    while (cin.read(buffer, TAM_BUFFER)) {
        // Se muestra la entrada
        cout.write(buffer, TAM_BUFFER);
    }

    // Lectura final
    cout.write(buffer, cin.gcount());
}
```

Copia de entrada en salida con **read** y **write** II

Resultado: ejemplo de uso (tecleando "Esto es una frase corta"+ ENTER) (24 caracteres). Hay dos vueltas de lectura de 10 caracteres y el bucle queda a la espera de recibir EOF (**Ctrl+D** en linux) o a tener caracteres suficientes para leer 10 caracteres. Tras pulsar **ENTER**) la salida obtenida es:

```
Esto es una frase co
```

Y finalmente, al introducir **EOF** aparece el resto de la frase teclada:

```
Esto es una frase corta
```

Índice

- 1 Introducción
- 2 Entrada/salida con formato
- 3 Entrada/salida sin formato
- 4 Estado de los flujos
 - Banderas de estado
 - Operaciones de consulta y modificación
 - Flujos en expresiones booleanas
- 5 Restricciones en el uso de flujos

Banderas de estado

Banderas de estado

Cada flujo mantiene un conjunto de banderas (*flags*) que indican si ha ocurrido un error en alguna operación de entrada/salida previa:

- eofbit: se activa cuando se encuentra el final de fichero (al recibir carácter EOF en una lectura).
- failbit: se activa cuando no se ha podido realizar una operación de E/S.
 - por ej., se intenta leer entero y se encuentra una letra
 - por ej., se intenta leer un carácter estando la entrada agotada
- badbit: se activa cuando ha ocurrido un error fatal (errores irrecurables).
- goodbit: está activada cuando ninguna de las otras banderas lo está

Operaciones de consulta y modificación

Hay una serie de métodos miembro para comprobar el estado del flujo, así como para cambiarlo explícitamente.

- `bool istream::good() const`: devuelve true si el flujo está bien (ninguno de los bits de error está activo)
- `bool istream::eof() const`: devuelve true si eofbit está activo
- `bool istream::fail() const`:
 - devuelve true si failbit o badbit está activo
 - si devuelve true fallarán la siguientes operaciones de lectura
- `bool istream::bad() const`: devuelve true si badbit está activo.
- `void istream::clear(iostate s=goodbit)`: limpia las banderas de error del flujo
- `void istream::setstate(iostate s)`: activa la bandera s
- `iostate istream::rdstate()`: devuelve las banderas de estado del flujo

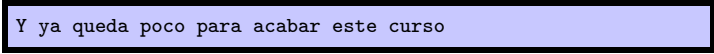
Ejemplo: eco de la entrada estándar I

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

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

    // Bucle de lectura
    while (!cin.eof()) {
        c=cin.get();
        cout.put(c);
    }
}
```

Resultado: ejemplo de uso (tecleando “Y ya queda poco para terminar este curso ”+ ENTER). Tras pulsar **ENTER** la salida obtenida es:



Y ya queda poco para acabar este curso

Después se pulsa Ctrl+D y aparece un símbolo extraño, al haber fallado la última lectura. ¿Cómo evitar este problema?

Ejemplo: eco de la entrada estándar I

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

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

    // Bucle de lectura
    while (!cin.eof()) {
        c=cin.get();

        // Solo se muestra el caracter si la lectura fue OK
        if(!cin.fail()){
            cout.put(c);
        }
    }
}
```

Resultado: ejemplo de uso (tecleando “Y ya queda poco para terminar este curso ”+ ENTER). Tras pulsar **ENTER** la salida obtenida es:

Ejemplo: eco de la entrada estándar II

Y ya queda poco para acabar este curso

Ejemplo: lectura de tres enteros I

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

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

    // Lectura del primer entero
    cin >> x;
    if (cin.fail()) cout << "Error 1" << endl;
    cout << x << endl;

    // Lectura del segundo entero
    cin >> x;
    if (!cin) cout << "Error 2" << endl;
    cout << x << endl;

    // Lectura del tercer entero
    cin >> x;
    if (!cin) cout << "Error 3" << endl;
    cout << x << endl;
}
```

Ejemplo: lectura de tres enteros II

Resultado: ejemplo de uso (tecleando "12 "+ ENTER, Hola + ENTER y 34 + ENTER):

```
12 // Entrada
12 // Primer valor leído
Hola // Segunda entrada
Error 2 // Salida provocada.....
0
Error 3
0
```

Ejemplo: lectura de tres enteros (mejorada) I

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

void procesa_error(const char *error) {
    // Se limpian los flags
    cin.clear();

    // Se muestra el error
    cout << error << endl;

    // Se elimina el resto de caracteres que quedaran en
    // la entrada
    while (cin.get() != '\n');
}
```

Ejemplo: lectura de tres enteros (mejorada) II

```
int main(int argc, char *argv[]) {  
    int x;  
  
    // Se lee el primer entero  
    cin >> x;  
    if (cin.fail()){  
        procesa_error("Error 1");  
    }  
    else {  
        cout << x << endl;  
    }  
  
    // Se lee el segundo entero  
    cin >> x;  
    if (!cin){  
        procesa_error("Error 2");  
    }  
    else{  
        cout << x << endl;  
    }  
}
```

Ejemplo: lectura de tres enteros (mejorada) III

```
// Se lee el tercer entero
cin >> x;
if (!cin){
    procesa_error("Error 3");
}
else{
    cout << x << endl;
}
}
```

Resultado: ejemplo de uso (tecleando "12 " + ENTER, Hola + ENTER y 34 + ENTER):

```
12 // Entrada
12 // Salida
Hola // Entrada erronea
Error 2 // Mensaje de error
34 // tercera entrada
34 // Salida
```


Flujos en expresiones booleanas

Podemos usar directamente un objeto **istream** u **ostream** en una expresión booleana.

- el valor del flujo es `true` si no se ha producido un error (equivalente a `!fail()`)
- también podemos usar el operador `!` con el flujo (equivalente a `fail()`).


Ejemplo: uso de flujos en expresiones booleanas I

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

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

    // Lectura de dato y comprobacion de error
    cin >> dato;
    if(cin)
        cout << "dato = " << dato << endl;
    else
        cout << "Error al leer entero" << endl;
}
```

Resultado: ejemplo de uso (tecleando "12" + ENTER):



```
12
dato = 12
```

Ejemplo: uso de flujos en expresiones booleanas II

Al teclear "10a" + ENTER:

```
10a  
dato = 10
```

Y al teclear "espacio+espacio+10a" + ENTER:

```
10a  
dato = 10
```

Finalmente, forzamos a que se produzca un error tecleando únicamente a:

```
a  
Error al leer entero
```

Flujos en expresiones booleanas

Puesto que la operación de lectura de un flujo devuelve el mismo flujo, podemos incluir la lectura en una expresión condicional.

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

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

    // Control del posible error
    if(cin >> dato)
        cout << "dato = " << dato << endl;
    else
        cout << "Error al leer entero" << endl;
}
```

Índice

- 1 Introducción
- 2 Entrada/salida con formato
- 3 Entrada/salida sin formato
- 4 Estado de los flujos
- 5 Restricciones en el uso de flujos

Restricciones:

- no podemos crear objetos `ostream` o `istream`.
- los flujos no tienen definidos ni **constructor de copia** ni `operator=`.
- por tanto, no es posible hacer asignaciones entre flujos
- tampoco es posible crear nuevos flujos mediante constructor de copia
- no podemos pasar un flujo como argumento por valor en una función
- los flujos tampoco deben pasarse como argumentos `const`
- un flujo debe ser pasado por referencia o mediante puntero

6 Flujos asociados a ficheros

- Introducción
- Tipos de ficheros
- Apertura y cierre de ficheros
- Modos de apertura de ficheros
- Ejemplos de programas con ficheros binarios y de texto
- Operaciones de posicionamiento
- Clase `fstream`

7 Flujos asociados a strings

Introducción

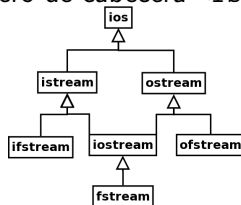
Conceptos básicos:

- un fichero es una secuencia de caracteres (bytes) almacenados en un dispositivo de almacenamiento masivo (disco duro, CD, ...)
- los ficheros permiten guardar los datos de forma *persistente*: serán accesibles por diferentes programas y en distintas ejecuciones
- para usar un fichero en C++ hemos de asociarle un flujo (según veremos más adelante) y trabajaremos con este flujo en la forma vista en las secciones anteriores:
 - operaciones de E/S con operadores << y >>
 - E/S con métodos de `istream` y `ostream` y funciones globales
 - manipuladores de formato
 - banderas de estado

Introducción

- en secciones anteriores se vio que mediante la redirección de E/S, un fichero podía usarse para leer o escribir
- eso hace que se asocie la entrada (`cin`) o salida estándar (`cout`) con un determinado fichero
- con este mecanismo, el nombre del fichero de E/S se fija en la línea de órdenes y no puede elegirse durante la ejecución del programa
- en esta sección veremos que podemos asociar un flujo con un fichero en tiempo de ejecución
- para usar ficheros incluiremos el fichero de cabecera `<fstream>`

- Jerarquía de clases:



Introducción I

- todo lo visto en secciones anteriores con `istream`, `ostream` y `iostream` se puede aplicar directamente a `ifstream`, `ofstream` y `fstream` respectivamente.
- si una función espera como parámetro un `istream` (`ostream`, `iostream`), podemos llamarla usando también un argumento de tipo `ifstream` (`ofstream`, `fstream`).

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

void escribe(ostream &f, string cad){
    f << cad;
}
```

Introducción II

```
int main(){
    string ca="Hola";

    // Nombre del archivo
    string nombreFichero="fichero.txt";

    // Apertura del archivo: se asocia el flujo correspondiente
    ofstream fich(nombreFichero.c_str());

    // Se usa el flujo
    escribe(fich,ca);

    // Se cierra el flujo
    fich.close();
}
```

Tipos de ficheros I

Ficheros de texto

en los ficheros de texto los datos se guardan en forma de caracteres imprimibles.

- eso hace que el número de caracteres usados al guardar un dato dependa del valor concreto del dato. Por ejemplo, si se guarda el valor `float` 123.4 mediante `"flujo << 123.4;"` se utilizarán 5 caracteres
- debería usarse algún separador entre los datos (Blanco, Enter, etc) para que los datos no estén mezclados y puedan leerse posteriormente
- es posible usar cualquier editor ASCII para ver o modificar el contenido de un fichero de texto
- la E/S se hace con los operadores `>>` y `<<` (y a veces con `get()`, `getline()` o `put()`)

Tipos de ficheros I

Ficheros binarios

en los ficheros binarios los datos se guardan usando directamente la representación interna de los datos en memoria: se transfiere el mismo contenido byte a byte entre memoria y fichero

- por ejemplo, si en nuestro ordenador se usa la representación de enteros con 4 bytes, usando complemento a dos, estos 4 bytes serán los que se usen para guardar cada entero en el fichero
- así, los datos que sean del mismo tipo, siempre ocupan la misma cantidad de memoria en el fichero. Esto permite conocer el lugar donde se encuentra un determinado dato dentro del fichero
- estos ficheros suelen ocupar menos espacio
- las operaciones de E/S son más eficientes ya que se pueden hacer lecturas/escrituras en bloque

Tipos de ficheros II

- como inconveniente, estos ficheros son menos portables entre plataformas distintas, ya que es posible que los datos se almacenen de forma distinta en ordenadores diferentes
- tampoco es posible ver o modificar el contenido de estos ficheros con un editor ASCII
- la E/S se debería hacer con los métodos: `read()`, `write()`, `get()`, `put()` (operaciones sin formato).
- no se usarán los operadores `>>` y `<<`.

Apertura y cierre de ficheros

Los pasos que hay que seguir para usar un fichero son:

- ❶ **apertura:** establece una asociación entre un flujo y un fichero de disco
 - implica que C++ prepare una serie de recursos para manejar el flujo, como creación de buffer, etc
- ❷ **transferencia de datos entre el programa y el fichero:** usaremos los operadores, métodos y funciones vistos en las secciones anteriores
- ❸ **cierre:** deshacer la asociación entre el flujo y el fichero de disco
 - implica que C++ descargue los buffers y libere los recursos que se crearon

Apertura de un fichero: `open()` I

```
void open(const char *filename, openmode mode);
```

Asocia el fichero `filename` al flujo y lo abre.

```
.....  
ifstream fi;  
ofstream fo;  
fstream fich;  
  
// Operaciones de apertura  
fi.open("ficheroDeEntrada.txt");  
fo.open("ficheroDeSalida.txt");  
fich.open("ficheroES.txt");  
.....
```


Apertura de un fichero: `open()` I

```
void open(const char *filename, openmode mode);
```

- el flujo queda preparado para realizar operaciones de E/S
- esta operación puede fallar. Por ejemplo, no podemos leer de un fichero que no exista o que no tenga permisos de lectura....
- el efecto de abrir un `ofstream` es que el fichero se crea para realizar salidas sobre él, y en caso de que ya exista, se vacía (se elimina su contenido previo)
- el parámetro `mode` es un parámetro por defecto cuyo valor depende del tipo de flujo (`ifstream`, `ofstream` o `fstream`)
- Este parámetro se utiliza para indicar el modo de apertura (lectura, escritura, binario, etc)

Apertura de un fichero con el constructor

Podemos usar un constructor del flujo para abrir el fichero al crear el flujo.

- `ifstream(const char *filename, openmode mode=in);`
- `ofstream(const char *filename, openmode mode=out);`
- `fstream(const char *filename, openmode mode=in|out);`

```
.....  
ifstream fi("ficheroDeEntrada.txt");  
ofstream fo("ficheroDeSalida.txt");  
fstream fich("ficheroES.txt");  
.....
```

Cierre de un fichero: `close()`

```
void close();
```

cierra el fichero eliminando la asociación entre el fichero y el flujo.

```
ifstream fi;  
ofstream fo;  
fstream fich;  
...  
fi.close();  
fo.close();  
fich.close();
```

- el objeto flujo no se destruye al ejecutar **close**
- podemos volver a asociar el flujo con otro fichero
- `close()` es llamado automáticamente por el destructor del flujo (cuando se destruye el objeto flujo), si el fichero está abierto

Ejemplo: lectura de archivo y muestra su contenido I

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

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

    // Se comprueba que se ha indicado el nombre del
    // archivo a leer
    if(argc==2) {
        ifstream f;

        // Apertura del archivo
        f.open(argv[1]);

        //comprobamos si se abrio correctamente
        if(f){
            char c;
```

Ejemplo: lectura de archivo y muestra su contenido II

```
// Bucle de lectura y salida de su contenido
while(f.get(c)){
    cout.put(c);
}

// Se cierra el archivo
f.close();
}
}
else{
    // Indicacion del error
    cerr<<"ERROR: no es posible abrir " << argv[1] << endl;
}
}
```

Resultado: ejemplo de uso cuando el contenido de **archivo.txt** es:

Ejemplo: lectura de archivo y muestra su contenido III

Contenido del archivo:
se mostrara directamente por pantalla
..... Y fin

```
Contenido del archivo:  
se mostrara directamente por pantalla  
..... Y fin
```

Ejemplo: copia de archivo I

```
// Fichero: mi_copy.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[]){
    // Forma de uso: nombre de archivo de origen y destino
    if(argc==3){
        // Apertura de archivo
        ifstream orig(argv[1]);

        // Control de posible error en la apertura
        if(!orig){
            cerr<<"Imposible abrir "<<argv[1]<<endl;
            return 1;
        }

        // Apertura del destino
        ofstream dest(argv[2]);
```

Ejemplo: copia de archivo II

```
// Control de posible error en apertura
if(!dest){
    cerr<<"Imposible abrir "<<argv[2]<<endl;
    return 1;
}

char c;

// Bucle de lectura
while(orig.get(c)){
    dest.put(c);
}

// No se llega a eof pero hay problema en dest
if(!orig.eof() || !dest){
    cerr<<"La copia no ha tenido exito"<<endl;
    return 1;
}
} // cierre y destruccion de orig, dest
return 0;
}
```


Ejemplo: copia de archivo III

Cuando el contenido de **archivo.txt** es el mostrado antes, se crea un nuevo archivo copia con el mismo contenido

```
Contenido del archivo:  
se mostrara directamente por pantalla  
..... Y fin
```

Modos de apertura de ficheros

Bandera	Significado
in	(input) apertura para lectura (modo por defecto en ifstream)
out	(output) apertura para escritura (modo por defecto en ofstream)
app	(append) la escritura en el fichero siempre se hace al final
ate	(at end) después de la apertura, se posiciona al final del archivo
trunc	(truncate) elimina los contenidos del fichero si ya existía
binary	la E/S se realiza en modo binario en lugar de texto

No todas las combinaciones tienen sentido. Algunas habituales son:

Banderas	Significado
in	apertura para lectura. Si el fichero no existe, falla
out	apertura para escritura. Si el fichero existe lo vacía. Si no existe lo crea
out app	apertura para añadir. Si el fichero no existe, lo crea
in out	apertura para lectura y escritura. Si el fichero no existe, falla la apertura
in out trunc	apertura para lectura y escritura. Si el fichero existe, lo vacía, y si no, lo crea

Además, podríamos usar:

- ate: después de la apertura nos situamos al final.
- binary: la E/S se hace en modo binario.

Ejemplo: escritura de archivo en binario con write() I

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

int main(int argc, char *argv[]){
    const int TAM=10;

    // Apertura del archivo en modo binario y para salida
    ofstream f("datos.dat",ios::out|ios::binary);

    // Si todo OK
    if(f){

        // Se escriben los valores del 0 al 9
        for(int i=0; i<TAM; ++i){
            f.write(reinterpret_cast<const char*>(&i), sizeof(int));
        }

        // Cierre del archivo
        f.close();
    }
}
```

Ejemplo: escritura de archivo en binario con write() II

```
else{  
    // Mensaje de error  
    cerr<<"Imposible crear datos.dat"<<endl;  
    return 1;  
}  
return 0;  
}
```

El archivo **datos.dat** no contiene caracteres imprimibles (formato binario).

Ejemplo: Escribimos en fichero binario con write() I

Ahora escribimos de una sola vez.

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

int main(int argc, char *argv[]){
    const int TAM=10;
    int data[TAM];

    // Apertura del archivo
    ofstream f("datos.dat",ios::out|ios::binary);

    // Control de posibles errores
    if(f){

        // Se almacenan los datos a guardar en data
        for(int i=0; i<TAM; ++i){
            data[i]=i;
        }
    }
}
```

Ejemplo: Escribimos en fichero binario con write() II

```
// Se escriben los datos de una sola vez
f.write(reinterpret_cast<const char*>(data), sizeof(int)*TAM);

// Se cierra el archivo
f.close();
}
else{
    // Control del posible error
    cerr<<"Imposible crear datos.dat"<<endl;
    return 1;
}
return 0;
}
```

Ejemplo: Leemos de fichero binario con read() I

Leemos de una sola vez.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[]){
    const int TAM=10;
    int data[TAM];

    // Apertura del archivo
    ifstream f("datos.dat",ios::in|ios::binary);

    // Control de posibles errores
    if(f){
        // Lectura de los datos
        f.read(reinterpret_cast<char*>(data),sizeof(int)*TAM);

        // Se cierra el archivo
        f.close();
    }
```

Ejemplo: Leemos de fichero binario con read() II

```
// Se muestran los datos leídos
for(int i=0;i<TAM;++i){
    cout<<data[i]<<" ";
}
cout<<endl;
}
else{
    // Control de errores
    cerr<<"Imposible abrir el fichero datos.dat"<<endl;
    return 1;
}
return 0;
}
```

La salida muestra todos los valores almacenados en el archivo.

Ejemplo: escritura de archivo de texto I

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

int main(int argc, char *argv[]){
    const int TAM=10;

    // Apertura del archivo
    ofstream f("datos.txt",ios::out);

    // Control de posibles errores
    if(f){
        // Bucle de escritura de datos
        for(int i=0; i<TAM; ++i){
            f<<i<<endl;
        }

        // Se cierra el archivo
        f.close();
    }
```

Ejemplo: escritura de archivo de texto II

```
else{  
    // Mensaje de error  
    cerr<<"Imposible crear datos.txt"<<endl;  
    return 1;  
}  
return 0;  
}
```

El contenido de **datos.txt** es:



0
1
2
3
4
5
6
7
8
9

Ejemplo: lectura de archivo de texto I

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[]){
    // Apertura del archivo
    ifstream f("datos.txt",ios::in);

    // Control de errores
    if(f){
        int i;

        // Bucle de lectura
        while(f>>i){
            cout<<i<<endl;
        }

        // Se cierra el archivo
        f.close();
    }

    else{
```

Ejemplo: lectura de archivo de texto II

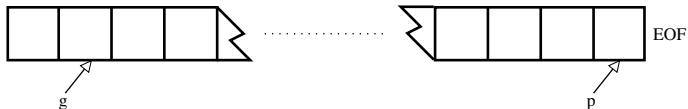
```
// Control de error
cerr<<"Imposible abrir el fichero datos.txt"<<endl;
return 1;
}
return 0;
}
```

Operaciones de posicionamiento

Punteros de posición

Cada flujo tiene asociados internamente dos *punteros de posición*: uno para lectura (**g**) y otro para escritura (**p**).

- cada uno apunta a la posición a la que toca leer o escribir a continuación con la siguiente operación de E/S
- cada vez que leemos o escribimos un carácter se avanza automáticamente el correspondiente puntero al siguiente carácter



Operaciones de posicionamiento

- Existen una serie de métodos en las clases `istream` y `ostream`, que permiten consultar y modificar los punteros de posición.

```
istream& istream::seekg(streamoff displ, ios::seekdir origen=ios::beg);
```

Cambia la posición del puntero de lectura

```
ostream& ostream::seekp(streamoff displ, ios::seekdir origen=ios::beg);
```

Cambia la posición del puntero de escritura

- Posibles valores para `origen`:

Valor	Desplazamiento relativo a
<code>ios::beg</code>	comienzo del flujo
<code>ios::cur</code>	posición actual
<code>ios::end</code>	final del flujo

- Ambos métodos devuelven `*this`.

Operaciones de posicionamiento I

```
streampos istream::tellg();
```

Devuelve la posición del puntero de lectura

```
streampos ostream::tellp();
```

Devuelve la posición del puntero de escritura

El ejemplo siguiente muestra cómo desplazar el puntero de lectura:

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[]){

    // Apertura del archivo
    ifstream f("datos.dat", ios::in|ios::binary);
```

Operaciones de posicionamiento II

```
// Control de posibles errores
if(f){
    int i,dato;

    // Se lee el numero de dato a leer
    cout<<"Numero de dato a leer: ";
    cin >> i;

    // Se desplaza el puntero de lectura
    f.seekg(i*sizeof(int));

    // Se lee el dato
    f.read(reinterpret_cast<char*>(&dato),sizeof(int));
    cout << "Dato leído: " << dato << endl;

    // Se cierra el archivo
    f.close();
}
```


Operaciones de posicionamiento III

```
else{  
    // Control de errores  
    cerr<<"Imposible abrir el fichero datos.dat"<<endl;  
    return 1;  
}  
return 0;  
}
```

Operaciones de posicionamiento I

Igual con el puntero de escritura:

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[]){
    ofstream f("datos.dat",ios::in|ios::out|ios::binary);
    if(f){
        int i,dato;
        cout<<"Numero de dato a modificar: ";
        cin>>i;
        cout<<"Nuevo dato (int): ";
        cin>>dato;
        f.seekp(i*sizeof(int));
        f.write(reinterpret_cast<char*>(&dato),sizeof(int));
        f.close();
    }
    else{
        cerr<<"Imposible abrir el fichero datos.dat"<<endl;
        return 1;
    }
}
```

Operaciones de posicionamiento II

```
}    return 0;
```

Clase fstream I

Permite hacer simultáneamente entrada y salida en un fichero, ya que deriva de ifstream y ofstream.

```
#include <fstream>
#include <iostream>
using namespace std;
int main() {
    // Apertura del archivo
    fstream fich("fstream.dat", ios::in | ios::out | ios::trunc | ios::binary);

    // Se lanza contenido al fichero
    fich << "abracadabra" << flush;

    // Se coloca el puntero de lectura al final
    fich.seekg(OL, ios::end);

    // Obtenemos la posición de lectura
    long lon = fich.tellg();
}
```

Clase fstream II

```
// Se coloca el puntero de lectura al inicio  
fich.seekg(0, ios::beg);
```

```
// Mientras haya datos por leer  
for(long i = 0L; i < lon; i++) {  
    // Si se encuentra una a, se cambia por una e  
    if(fich.get() == 'a') {  
        fich.seekp(i, ios::beg);  
        fich << 'e';  
    }  
}
```

```
// Se muestra la salida obtenida  
cout << "Salida: ";  
fich.seekg(0L, ios::beg);  
for(long i = 0L; i < lon; i++){  
    cout << (char)fich.get();  
}  
cout << endl;  
fich.close();  
}
```

Clase fstream III

La salida obtenida, como resultado de cambiar en **abracadabra** los caracteres **a** por **e**:

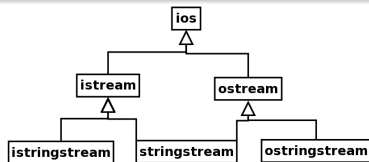
```
Salida: ebrecedebre
```

6 Flujos asociados a ficheros

7 Flujos asociados a strings

Flujos asociados a strings I

Podemos crear flujos de E/S en los que el fuente o destino de los datos es un objeto `string`. O sea, podemos manejar el tipo `string` como fuente o destino de datos de un flujo.



Incluiremos el fichero de cabecera `<sstream>` para usar estas clases.

- `istringstream`: flujo de entrada a partir de un `string`
- `ostringstream`: flujo de salida hacia un `string`
- `stringstream`: flujo de E/S con un `string`

Funciones miembro I

str()

- `string str() const;`
 - Obtiene una copia del objeto `string` asociado al flujo.
- `void str(const string &s);`
 - Copia el contenido del `string s` al `string` asociado al flujo.

```

1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 using namespace std;
5 int main() {
6     int val,n;
7     istringstream iss;
8     string strvalues = "32 240 2 1450";
9     iss.str (strvalues);
10    for (n=0; n<4; n++){
11        iss >> val;
12        cout << val+1 << endl;
13    }
14    return 0;
15 }
```



```

33
241
3
1451
```

Ejemplos

Ejemplo

Este ejemplo muestra cómo un `ostringstream` puede usarse para convertir cualquier tipo de dato a un `string`.

```
1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4 int main() {
5     ostringstream f;
6     f<<15*15; // almacenamos un int en el flujo
7     string s=f.str();
8     cout<<"15 x 15 = "<<s<<endl;
9 }
```



15 x 15 = 225



Ejemplos

Ejemplo

Este ejemplo muestra cómo un `istringstream` puede usarse para convertir datos guardados en un `string` a cualquier tipo de dato.

```

1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4 int main() {
5     istringstream flujo;
6     flujo.str("15.8 true 12");
7     float f;
8     bool b;
9     int i;
10
11     flujo >> f >> boolalpha >> b >> i;
12     cout << "f = " << f
13         << boolalpha << "\nb = " << b
14         << "\ni = " << i << endl;
15 }
```



```
f = 15.8
b = true
i = 12
```



Ejemplos

Ejemplo

Ejemplo similar al anterior, muestra cómo un stringstream puede usarse para convertir datos guardados en un string a cualquier tipo de dato.

```
1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4 int main() {
5     stringstream flujo;
6     float f;
7     bool b;
8     int i;
9
10    flujo << "15.8 true 12";
11    flujo >> f >> boolalpha >> b >> i;
12    cout << "f = " << f
13          << boolalpha << "\nb = " << b
14          << "\ni = " << i << endl;
15 }
```



```
f = 15.8
b = true
i = 12
```



Inicialización con los constructores

Constructor

A parte de los constructores por defecto, los flujos basados en string disponen de constructores que permiten inicializar el string asociado al flujo.

- `istringstream(openmode modo=ios::in);`
- `istringstream(const string &str, openmode modo=ios::in);`
- `ostringstream(openmode modo=ios::out);`
- `ostringstream(const string &str, openmode modo=ios::in|ios::out);`
- `stringstream(openmode modo=ios::out);`
- `stringstream(const string &str, openmode modo=ios::in|ios::out);`

Ejemplo de uso de constructor

Ejemplo

Ejemplo similar a los anteriores, muestra cómo un `istringstream` puede usarse para convertir datos guardados en un `string` a cualquier tipo de dato. Los datos son insertados en el `istringstream` con el constructor.

```

1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4 int main() {
5     istringstream flujo("15.8 true 12");
6     float f;
7     bool b;
8     int i;
9     flujo >> f >> boolalpha >> b >> i;
10    cout << "f = " << f
11         << boolalpha << "\nb = " << b
12         << "\ni = " << i << endl;
13 }
```



```
f = 15.8
b = true
i = 12
```



¡Eso es todo amigos!

Mucha suerte

con todos vuestros exámenes, y especialmente con el de Metodología de la Programación.

