

Metodología de la Programación

Tema 6. Gestión de E/S. Ficheros

Departamento de Ciencias de la Computación e I.A.



ETSIIT Universidad de Granada

Curso 2011-12

Contenido del tema

1 Introducción

- Flujos de E/S
- Clases y ficheros de cabecera
- Flujos y búferes

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

- Flujos asociados a ficheros
- Introducción
- Tipos de ficheros: texto y binarios
- 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

Contenido del tema

1 Introducción

- Flujos de E/S
- Clases y ficheros de cabecera
- Flujos y búferes

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

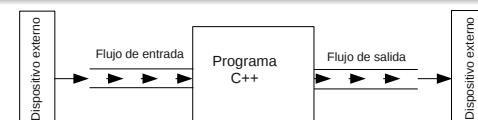
- Flujos asociados a ficheros
- Introducción
- Tipos de ficheros: texto y binarios
- Apertura y cierre de ficheros
- Modos de apertura de ficheros
- Ejemplos de programas con ficheros binarios y de texto
- Operaciones de posicionamiento
- Clase fstream

7 Flujos asociados a strings

Flujos de E/S

Flujo (stream)

Es una *abstracción* que representa cualquier fuente o consumidor de datos y que permite realizar operaciones de E/S de datos con él (enviar o recibir datos). Podemos ver un flujo como una secuencia de bytes que fluye desde o hacia algún dispositivo.



- 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 otra cosa.

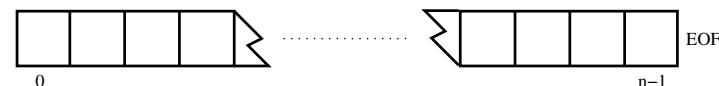
Tipos de flujos

- 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, etc).
- Flujos de entrada/salida:** La secuencia de bytes puede fluir en ambos sentidos.

Tamaño finito de los flujos

Podemos ver un flujo de datos como una secuencia de n caracteres consecutivos.

- Una vez leídos los n caracteres, la lectura de un nuevo carácter implica un error que impide que se sigan leyendo caracteres.
- Si usamos por ejemplo `get()` para leer, se devolverá la constante especial `EOF` (*end of file*) cuando se intenta leer después de leer los n caracteres.
- Así, podemos ver el flujo como una secuencia de n caracteres, seguida por la constante `EOF`.



Flujos estándar

- La inclusión del fichero de cabecera `<iostream>` nos permite disponer de varios flujos para la E/S estándar.
- Cada flujo tiene asociado un dispositivo físico sobre el que actúa.
- Flujos estándar predefinidos:**
 - `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 búfer (pantalla).
 - `clog`: Instancia de `ostream` conectado a la salida estándar de error (pantalla).

Redireccionamiento de entrada/salida

Las *shells* de los sistemas operativos proporcionan mecanismos para cambiar la entrada, salida estándar o de error.

Redirección de salida

programa > salida.txt

Redirección de entrada

programa < entrada.txt

Redirección de salida de error

programa >& error.txt

Redirigiéndolo todo

(programa < entrada.txt > salida.txt) >& fichero.txt

Encauzamiento

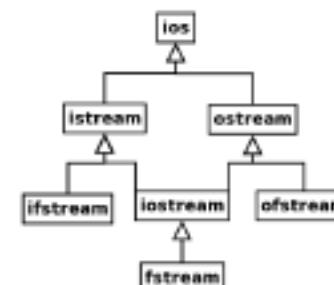
programa1 < entrada1.txt | programa2 | programa3 > salida3.txt

Ficheros de cabecera

Todo lo que se define en estos ficheros está incluido en el *namespace std*.

- <iostream>: 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 <iostream> 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.

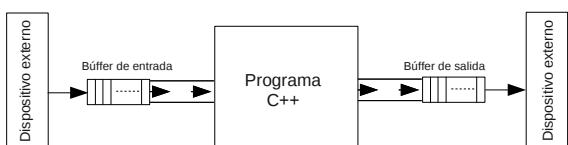
Jerarquía de clases



- *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 búferes

- 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 **búfer intermedio** para almacenamiento temporal de los datos.
- Cuando el búfer se llena, se hace la transferencia a o desde el dispositivo.
- El método *ostream::flush()* o bien *endl* ordenan la transferencia inmediata de un búfer de salida al dispositivo.



Flujos y búferes

- Al hacer *cout* de algo no aparecerá en pantalla hasta que se llene el búfer o usemos *ostream::flush()* o *endl*.

```

int main(){
    cout<< "Prueba";
    while(true);
}
  
```



```

int main(){
    cout<< "Prueba";
    cout.flush();
    while(true);
}
  
```



```

int main(){
    "Prueba" << endl;
    while(true);
}
  
```



- *cerr* es un flujo que no espera a que se llene su búfer para transferir los datos.

```

int main(){
    cerr<< "Prueba";
    while(true);
}
  
```



Contenido del tema

- 1 Introducción
 - Flujos de E/S
 - Clases y ficheros de cabecera
 - Flujos y búferes
- 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()

Introducción a entrada/salida con formato

```

1 #include <iostream>
2 using namespace std;
3 int main(){
4     const char* S1 = "AEIO";
5     float S2 = 12.4;
6
7     cout.write(S1, 4);
8     cout.write(reinterpret_cast<const char*>(&S2),
9                 sizeof(float)); //salida sin formato
10    cout << endl;
11    cout << S1 << S2 << endl; //salida con formato
12 }
```



AEIOFFFA
AEIO12.4

Introducción a entrada/salida con formato

E/S con formato

- Implica una transformación de los caracteres (bytes) que se leen/escriben al tipo de dato usado para leer o escribir.
- La transformación se hace entre la representación interna de los datos y una representación comprensible por los usuarios (secuencia de caracteres imprimibles).

Por ejemplo al mostrar un dato double en pantalla con cout se transforma la representación interna de un dato de este tipo en caracteres imprimibles en pantalla.

- 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.

operator <<

- 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 dependiendo del tipo de la variable dato.
- ```
cout << dato; // los datos fluyen en la dirección de las flechas
```
- 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 devuelve una referencia al mismo objeto ostream que hemos usado para llamarlo. 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.

## operator <<: Ejemplo

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main()
5 {
6 string cads="Hola";
7 int x=74;
8 double y=65.1234;
9 int *ptr=&x;
10 bool valor=false;
11 const char *adios = "Adios";
12 char cadena[100]="Cadena";
13 cout << "cads: " << cads << endl;
14 cout << "x: " << x << " y: " << y << endl;
15 cout << "ptr: " << ptr << " valor: " << valor << endl;
16 cout << "adios: " << adios << endl;
17 cout << static_cast<const void *>(adios) << endl;
18 cout << "cadena: " << cadena << endl;
19 }

```



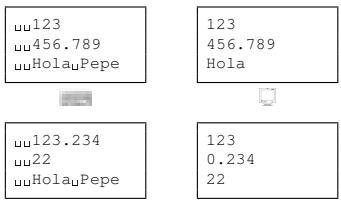
## operator >>: Ejemplo

- 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.

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main()
5 {
6 char cad[100];
7 int x;
8 float y;
9
10 cin >> x >> y;
11 cin >> cad;
12 cout << x << endl << y << endl;
13 cout << cad << endl;
14 }

```



## operator >>

- Se conoce como el **operador de extracción de flujos**.
  - Cuando el compilador de C++ encuentra una expresión como la siguiente, busca la versión del método a llamar dependiendo del tipo de la variable dato.
- cin >> dato; // los datos fluyen en la dirección de las flechas*
- 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 devuelve una referencia al mismo objeto istream que hemos usado para llamarlo. Esto permite encadenar entradas.
  - 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.

## Banderas de formato |

- Cada flujo tiene asociadas una serie de banderas (indicadores) de formato para controlar la apariencia de los datos que se escriben o se leen.
- Son variables miembro enum de tipo fmtflags (long int) definidas en la clase ios\_base (superclase de ios).

```

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 II

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

## Modificación del formato

```

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 cout.setf(ios::scientific,ios::floatfield);
6 cout << 123.45 << endl;
7 cout.setf(ios::fixed,ios::floatfield);
8 cout << 123.45 << endl;
9 }

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 cout << 123 << endl;
6 cout.setf(ios::showpos);
7 cout << 123 << endl;
8 }

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 cout << 123 << endl;
6 cout.setf(ios::hex,ios::basefield);
7 cout << 123 << endl;
8 cout.setf(ios::showbase);
9 cout << 123 << endl;
10 cout.setf(ios::oct,ios::basefield);
11 cout << 123 << endl;
12 cout.setf(ios::fmtflags(0),ios::showbase);
13 cout << 123 << endl;
14 }

```



```
1.234500e+02
123.450000
```



```
123
+123
```



```
123
7b
0x7b
0173
173
```

## Modificación del formato

La clase `ios_base` dispone de métodos para modificar o consultar las banderas de formato.

- `fmtflags setf(fmtflags banderas):`

- Para activar banderas del flujo.
- `banderas` es un conjunto de una o más banderas unidas con el operador OR lógico a nivel de bits.
- Devuelve el estado de las banderas anterior al cambio.

- `fmtflags setf(fmtflags banderas, fmtflags mask):`

- Se usa para activar una de las banderas de un grupo, usando una máscara en el parámetro `mask`.

| banderas                            | mask                     |
|-------------------------------------|--------------------------|
| <code>left, right o internal</code> | <code>adjustfield</code> |
| <code>dec, oct o hex</code>         | <code>basefield</code>   |
| <code>scientific o fixed</code>     | <code>floatfield</code>  |

- Devuelve el estado de las banderas anterior al cambio.

## Modificación del formato

- `void unsetf(fmtflags banderas)`

- Sirve para desactivar banderas.
- `banderas` es un conjunto de una o más banderas unidas con el operador `|`.

- `fmtflags flags() const`

- Devuelve las banderas del flujo.

- `fmtflags flags(fmtfl) const`

- Establece nuevas banderas en el flujo, borrando todas las que hubiera anteriormente.

- Devuelve las banderas anteriores al cambio.

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5 cout.flags (ios::right | ios::hex | ios::showbase);
6 cout.width (10);
7 cout << 100;
8 return 0;
9 }

```



```
uuuuuu0x64
```

## Modificación del formato

- **precision()**
  - **streamsize precision() const:** Devuelve la precisión (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.
- Los tres métodos que modifican un valor devuelven el valor que tenía antes de la modificación.

## 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.

| 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)                  |       |

## Modificación del formato

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5 cout.width(20);
6 cout.fill('.');
7 cout.setf(ios::right,ios::adjustfield);
8 cout << 123.45 << endl;
9 cout << 123.45 << endl;
10 cout.width(10);
11 cout << 123.45 << endl;
12 }
```



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

## Manipuladores de formato

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4 int main(int argc, char *argv[])
5 {
6 cout << setbase(16) << showbase << 20 << endl;
7 cout << hex << 20 << endl;
8 cout << oct << noshowbase << 20 << endl;
9 cout << dec << 0x20 << endl;
10 cout << setprecision(3) << 2.123456 << endl;
11 cout << setw(10) << 2.123456 << endl;
12 cout << setw(10) << left << 2.123456 << endl;
13 cout << setw(10) << left << setfill('*') << 2.123456 << endl;
14 cout << setw(10) << right << setfill('*') << 2.123456 << endl;
15 }
```



```
0x14
0x14
24
32
2.12
uuuuuu2.12
2.12
2.12*****
*****2.12
```

## Manipuladores de formato

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6 int integerValue = 1000;
7 double doubleValue = 0.0947628;
8
9 cout << "Valor banderas: " << cout.flags()
10 << "\nint y double en formato original:\n"
11 << integerValue << '\t'
12 << doubleValue << endl << endl;
13
14 ios_base::fmtflags originalFormat = cout.flags();
15 cout << showbase << oct << scientific;
16
17 cout << "Valor banderas: " << cout.flags()
18 << "\nint y double en nuevo formato:\n"
19 << integerValue << '\t'
20 << doubleValue << endl << endl;
21
22 cout.flags(originalFormat); // restaurar formato
23
24 cout << "Valor restaurado de banderas: "
25 << cout.flags()
26 << "\nValores en formato original:\n"
27 << integerValue << '\t'
28 << doubleValue << endl;
29 } // end main

```



```

Valor banderas: 4098
int y double en formato original:
1000 0.0947628

Valor banderas: 011500
int y double en nuevo formato:
01750 9.476280e-02

Valor restaurado de banderas: 4098
Valores en formato original:
1000 0.0947628

```



## Contenido del tema

- 1 Introducción
  - Flujos de E/S
  - Clases y ficheros de cabecera
  - Flujos y búferes
- 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
- 6 Flujos asociados a ficheros
  - Introducción
  - Tipos de ficheros: texto y binarios
  - 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

## ostream::put()

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

- Envía el carácter c al objeto ostream que lo llama.
- Devuelve una referencia al flujo que lo llama.

```

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 char c1='a', c2='b';
6 cout.put(c1);
7 cout.put(c2);
8 cout.put('c');
9 cout.put('\n');
10 cout.put(c1).put(c2).put('c').put('\n');
11 cout << c1 << c2 << 'c' << '\n';
12 }

```



```

abc
abc
abc

```



## ostream::write()

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

- Envía al objeto ostream que lo llama, el bloque de datos apuntados por s, con un tamaño de n caracteres.
- Devuelve una referencia al flujo que lo llama (\*this).
- Suele usarse con flujos asociados a ficheros y no con cout.

```

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 const char *cad="Hola";
6 int x=0x414243;
7 cout.write(cad,4);
8 cout.write("Adios\n",6);
9 cout.write((char*)&x,1);
10 cout.write(((char*)&x)+1,2);
11 }

```



```

HolaAdios
CBA

```

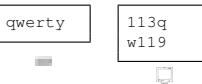


## istream::get()

```
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 del 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.

```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[]){
4 int ci; char cc;
5 ci = cin.get();
6 cout << ci << (char)ci << endl;
7 cc = cin.get();
8 cout << cc << (int)cc << endl;
9 }
```



## istream::get()

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

- Extrae un carácter del flujo y lo almacena en c.
- Devuelve una referencia al objeto istream que lo llamó (\*this).

```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[]){
4 char c1,c2,c3;
5 cin.get(c1);
6 cin.get(c2);
7 cin.get(c3);
8 cout << c1 << c2 << c3 << endl;
10 }
```



qwerty

qwe

```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[]){
4 char c1,c2,c3;
5 cin.get(c1).get(c2).get(c3);
6 cout << c1 << c2 << c3 << endl;
8 }
```



qwerty

qwe

## 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:
  - Hayamos leído  $n - 1$  caracteres.
  - O hayamos encontrado el carácter delim.
  - O 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 que lo llamó (\*this).

## istream::get()

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5 char c1[50],
6 c2[50],
7 c3[50];
8 cin.get(c1,50);
9 cin.get(c2,50,'a');
10 cin.get(c3,50,'t');
11 cout << c1 << endl;
12 cout << c2 << endl;
13 cout << c3 << endl;
14 }
```

Hola como estas  
Hola como estasHola como estas  
Hol  
a como es

## istream::getline()

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

- Es 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:
  - Hayamos leído  $n - 1$  caracteres.
  - O hayamos encontrado el carácter delim.
  - O hayamos llegado al final del flujo o encontrado 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 que lo llamó (\*this).

## Función global 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.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main(){
5 string cad1,cad2;
6 getline(cin,cad1);
7 cout << cad1 << endl;
8 getline(cin,cad2,'m');
9 cout << cad2 << endl;
10 }
```



Hola como estas  
Hola como estas

Hola como estas  
Hola co

## istream::getline()

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5 char c1[50],
6 c2[50],
7 c3[50];
8 cin.getline(c1,50);
9 cin.getline(c2,50,'a');
10 cin.getline(c3,50,'t');
11 cout << c1 << endl;
12 cout << c2 << endl;
13 cout << c3 << endl;
14 }
```



Hola como estas  
Hola como estas

Hola como estas  
Hol  
como es

## istream::ignore()

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

- Extrae caracteres del flujo y no los almacena en ningún sitio.
  - Hasta que hayamos leído  $n$  caracteres.
  - O hayamos encontrado el carácter delim. En este caso delim también es extraido.
- Devuelve una referencia al objeto istream que lo llamó (\*this).

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main()
5 {
6 string c;
7 cin.ignore(6);
8 getline(cin,c);
9 cout << c << endl;
10 }
```



1234567890

7890

## istream::read() y istream::readsome()

```
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 se encuentra EOF, se guardan en el array los caracteres leídos hasta ahora y se activan los bits failbit y eofbit.
- Devuelve una referencia al objeto *istream* que lo llamó (*\*this*).
- *s* debe tener reservada suficiente memoria.
- No añade un carácter '\0' al final de *s*.

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

- Tiene la misma función que *read()* pero ésta devuelve el número de caracteres extraídos con éxito.

## Devolución de datos al flujo

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main()
5 {
6 unsigned char c;
7 c = cin.get();
8 cin.unget();
9 if (isdigit(c))
10 {
11 int n;
12 cout << "Es un numero" << endl;
13 cin >> n;
14 cout << n << endl;
15 }
16 else
17 {
18 string cad;
19 cout << "Es una cadena" << endl;
20 cin >> cad;
21 cout << cad << endl;
22 }
23 }
```



1234

Es un numero

1234

## Consultas al flujo

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main(int argc, char *argv[])
5 {
6 if (isdigit(cin.peek()))
7 {
8 int n;
9 cout << "Es un numero" << endl;
10 cin >> n;
11 cout << n << endl;
12 }
13 else
14 {
15 string cad;
16 cout << "Es una cadena" << endl;
17 cin >> cad;
18 cout << cad << endl;
19 }
20 }
```



1234

Es un numero  
1234

Hola

Es una cadena  
Hola

## Ejemplo con read(): Cálculo tamaño de un fichero

```

1 #include <iostream>
2 using namespace std;
3 int main ()
4 {
5 const int TAM_BUFFER = 10;
6 char buffer[TAM_BUFFER];
7 int tam = 0;
8 while (cin.read(buffer, TAM_BUFFER)){
9 tam += TAM_BUFFER;
10 }
11 tam += cin.gcount();
12 cout << "Tamanio = " << tam << endl;
13 }
```



Este es un ejemplo para calcular el tamaño

Tamaño = 32

## Ejemplo con read() y write(): Copia de entrada en salida

```

1 #include <iostream>
2 using namespace std;
3 int main ()
4 {
5 const int TAM_BUFFER = 10;
6 char buffer[TAM_BUFFER];
7 while (cin.read(buffer, TAM_BUFFER))
8 {
9 cout.write(buffer, TAM_BUFFER);
10 }
11 cout.write(buffer, cin.gcount());
12 }
```



Este es un ejemplo de copia de la entrada

Este es un ejemplo de copia de la entrada

## Contenido del tema

- 1 Introducción
  - Flujos de E/S
  - Clases y ficheros de cabecera
  - Flujos y búferes
- 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
- 6 Flujos asociados a ficheros
  - Introducción
  - Tipos de ficheros: texto y binarios
  - 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

## Banderas de estado

### Banderas de estado

Cada flujo mantiene un conjunto de banderas (*flags*) que indican si ha ocurrido un error en una 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 irrecuperables).
- **goodbit**: Está activada cuando ninguna de las otras 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 las siguientes operaciones de lectura que hagamos.
- **bool istream::bad() const**: Devuelve 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

```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 int c;
6 while (!cin.eof())
7 {
8 c=cin.get();
9 cout.put(c);
10 }
11 }

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 int c;
6 while (!cin.eof())
7 {
8 c=cin.get();
9 if(!cin.fail())
10 cout.put(c);
11 }
12 }
```

## Ejemplo: Lectura de tres enteros

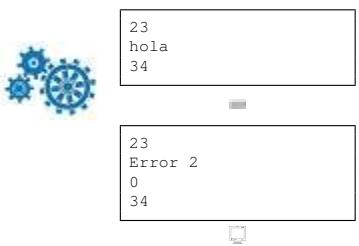
```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[])
4 {
5 int x;
6 cin >> x;
7 if (cin.fail()) cout << "Error 1" << endl;
8 cout << x << endl;
9 cin >> x;
10 if (!cin) cout << "Error 2" << endl;
11 cout << x << endl;
12 cin >> x;
13 if (!cin) cout << "Error 3" << endl;
14 cout << x << endl;
15 }
```

## Ejemplo: Lectura de tres enteros

```

1 #include <iostream>
2 using namespace std;
3 void procesa_error(const char *error) {
4 cin.clear();
5 cout << error << endl;
6 while (cin.get() != '\n');
7 }
8 int main(int argc, char *argv[]) {
9 int x;
10 cin >> x;
11 if (cin.fail()) procesa_error("Error 1");
12 cout << x << endl;
13 cin >> x;
14 if (!cin) procesa_error("Error 2");
15 cout << x << endl;
16 cin >> x;
17 if (!cin) procesa_error("Error 3");
18 cout << x << endl;
19 }

```



## Flujos en expresiones booleanas

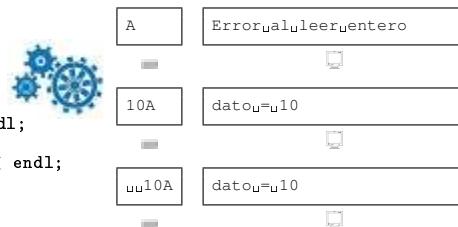
Podemos usar directamente un objeto `istream` u `ostream` en una expresión booleana.

- 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()`).

```

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[]) {
4 int dato;
5 cin >> dato;
6 if(cin)
7 cout << "dato = " << dato << endl;
8 else
9 cout << "Error al leer entero" << endl;
10 }

```



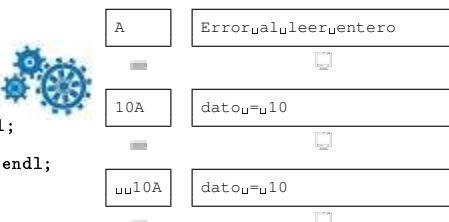
## Flujos en expresiones booleanas

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

```

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[]) {
4 int dato;
5
6 if(cin >> dato)
7 cout << "dato = " << dato << endl;
8 else
9 cout << "Error al leer entero" << endl;
10 }

```



## Contenido del tema

- 1 Introducción
  - Flujos de E/S
  - Clases y ficheros de cabecera
  - Flujos y búferes
- 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
  - Flujos asociados a ficheros
    - Introducción
    - Tipos de ficheros: texto y binarios
    - Apertura y cierre de ficheros
    - Modos de apertura de ficheros
    - Ejemplos de programas con ficheros binarios y de texto
    - Operaciones de posicionamiento
    - Clase `fstream`
  - Flujos asociados a strings

## Restricciones en el uso de flujos

- 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 bien como un puntero al flujo.

## Contenido del tema

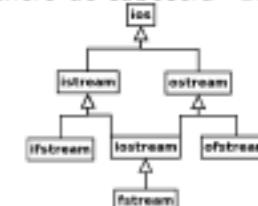
- 1 Introducción
  - Flujos de E/S
  - Clases y ficheros de cabecera
  - Flujos y búferes
- 2 Entrada/salida con formato
  - Introducción
  - Banderas de formato
  - Modificación del formato
  - Manipuladores de formato
- 3 Entrada/salida sin formato
  - Salida sin formato
  - Entrada sin formato
  - 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
- 6 Flujos asociados a ficheros
  - Introducción
  - Tipos de ficheros: texto y binarios
  - 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

- 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* de forma que sean accesibles por diferentes programas y ejecuciones.
- Para usar un fichero en C++, le asociaremos un flujo (según veremos más adelante) y trabajaremos con este flujo en la forma que hemos visto 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, hemos visto que mediante la redirección de E/S, un fichero podía usarse para leer o escribir.
- Eso hacía que se asociase 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

- Todo lo que hemos 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`).

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 using namespace std;
5 void escribe(ostream &f, string cad){
6 f << cad;
7 }
8 int main(){
9 string ca="Hola";
10 string nombreFichero="fichero.txt";
11 ofstream fich(nombreFichero.c_str());
12 escribe(fich,ca);
13 fich.close();
14 }
```



## Tipos de ficheros: texto y binarios 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: texto y binarios II

### Ficheros binarios

En los ficheros binarios los datos se guardan usando directamente la representación que tienen los datos internamente 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 en el 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: texto y binarios III

- 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()`.
- No usar los operadores `>>` y `<<`.

## Apertura y cierre de ficheros

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

- ① **Abrir el fichero:** Establece una asociación entre un flujo y un fichero de disco.
  - Internamente implica que C++准备 una serie de recursos para manejar el flujo, como creación de búferes.
- ② **Transferir datos entre el programa y el fichero:** Usaremos los operadores, métodos y funciones vistos en las secciones anteriores.
- ③ **Cerrar el fichero:** Deshacer la asociación entre el flujo y el fichero de disco.
  - Internamente implica que C++ descargue los búferes y libere los recursos que se crearon.

## Apertura de un fichero: open()

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

Asocia el fichero filename al flujo y lo abre.

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

- El flujo queda preparado para realizar operaciones de E/S.
- Esta operación puede fallar. Por ejemplo, no podemos leer de un fichero que no existe.
- 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.
- El parámetro mode es un parámetro por defecto cuyo valor por defecto 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 cerrar el flujo.
- 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 open() y close()

```

1 // Fichero: mi_cat.cpp
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5 int main(int argc, char *argv[])
6 {
7 if(argc==2) {
8 ifstream f;
9 f.open(argv[1]);
10 if(f){ //comprobamos si se abrio correctamente
11 char c;
12 while(f.get(c)){
13 cout.put(c);
14 }
15 }
16 f.close();
17 } else
18 cerr<<"ERROR. No es posible abrir "
19 <<argv[1]<<endl;
20 }

```

```
>cat mi_cat.txt
Esto mismo va a salir
por pantalla
```

```
>./mi_cat mi_cat.txt
Esto mismo va a salir
por pantalla
```

## 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) Despues 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 apertura/cierre con constructor/destructor

```
>mi_copy ficheroE.txt ficheroS.txt
```

```

1 // Fichero: mi_copy.cpp
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5 int main(int argc, char *argv[]){
6 if(argc==3){
7 ifstream orig(argv[1]); //Creacion y apertura
8 if(!orig){ // equivalente a if(orig.fail())
9 cerr<<"Imposible abrir "<<argv[1]<<endl;
10 return 1;
11 }
12 ofstream dest(argv[2]); //Creacion y apertura
13 if(!dest){ // equivalente a if(dest.fail())
14 cerr<<"Imposible abrir "<<argv[2]<<endl;
15 return 1;
16 }
17 char c;
18 while(orig.get(c))
19 dest.put(c);
20 if(!orig.eof() || !dest){
21 cerr<<"La copia no ha tenido exito"<<endl;
22 return 1;
23 }
24 } //cierra y destrucción de orig, dest
25 return 0;
26 }

```

## Ejemplo: Escribimos en fichero binario con write()



```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(int argc, char *argv[]){
5 const int TAM=10;
6 ofstream f("datos.dat",ios::out|ios::binary);
7 if(f){
8 for(int i=0; i<TAM; ++i){
9 f.write(reinterpret_cast<const char*>(&i),
10 sizeof(int));
11 }
12 f.close();
13 }
14 else{
15 cerr<<"Imposible crear datos.dat"<<endl;
16 return 1;
17 }
18 return 0;
19 }

```

## Ejemplo: Escribimos en fichero binario con write()

Ahora escribimos de una sola vez.

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(int argc, char *argv[]){
5 const int TAM=10;
6 int data[TAM];
7 ofstream f("datos.dat",ios::out|ios::binary);
8 if(f){
9 for(int i=0; i<TAM; ++i){
10 data[i]=i;
11 }
12 f.write(reinterpret_cast<const char*>(data),sizeof(int)*TAM);
13 f.close();
14 }
15 else{
16 cerr<<"Imposible crear datos.dat"=<<endl;
17 return 1;
18 }
19 return 0;
20 }
```



## Ejemplo: Escribimos en fichero de texto

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(int argc, char *argv[]){
5 const int TAM=10;
6 ofstream f("datos.txt",ios::out);
7 if(f){
8 for(int i=0; i<TAM; ++i){
9 f<<i<<endl;
10 }
11 f.close();
12 }
13 else{
14 cerr<<"Imposible crear datos.txt"=<<endl;
15 return 1;
16 }
17 return 0;
18 }
```

>cat datos.txt

|   |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |



## Ejemplo: Leemos de fichero binario con read()

Leemos de una sola vez.

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(int argc, char *argv[]){
5 const int TAM=10;
6 int data[TAM];
7 ifstream f("datos.dat",ios::in|ios::binary);
8 if(f){
9 f.read(reinterpret_cast<char*>(data),sizeof(int)*TAM);
10 f.close();
11 for(int i=0;i<TAM;++i){
12 cout<<data[i]<<" ";
13 }
14 cout<<endl;
15 }
16 else{
17 cerr<<"Imposible abrir el fichero datos.dat"=<<endl;
18 return 1;
19 }
20 return 0;
21 }
```



## Ejemplo: Leemos de fichero de texto

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(int argc, char *argv[]){
5 ifstream f("datos.txt",ios::in);
6 if(f){
7 int i;
8 while(f>>i){
9 cout<<i<<endl;
10 }
11 f.close();
12 }
13 else{
14 cerr<<"Imposible abrir el fichero datos.txt"=<<endl;
15 return 1;
16 }
17 return 0;
18 }
```

|   |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |

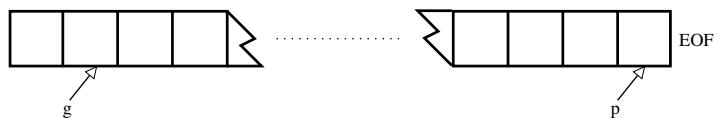


## Operaciones de posicionamiento

### 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

**streampos istream::tellg();**

Devuelve la posición del puntero de lectura

**streampos ostream::tellp();**

Devuelve la posición del puntero de escritura

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(int argc, char *argv[]){
5 ofstream f("datos.dat",ios::out|ios::binary);
6 if(f){
7 int i,dato;
8 cout<<"Número de dato a modificar: ";
9 cin>>i;
10 cout<<"Nuevo dato (int): ";
11 cin>>dato;
12 f.seekp(i*sizeof(int));
13 f.write(reinterpret_cast<char*>(&dato),sizeof(int));
14 f.close();
15 }
}
```



## 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 despl, ios::seekdir origen=ios::beg);**

Cambia la posición del puntero de lectura

**ostream& ostream::seekg(streamoff despl, ios::seekdir origen=ios::beg);**

Cambia la posición del puntero de escritura

- Posibles valores para origen:

| Valor    | Desplazamiento relativo a |
|----------|---------------------------|
| ios::beg | Comienzo del flujo        |
| ios::cur | Posición actual           |
| ios::end | Final del flujo           |

- Ambos métodos devuelven **\*this**.

## Operaciones de posicionamiento

**streampos istream::tellg();**

Devuelve la posición del puntero de lectura

**streampos ostream::tellp();**

Devuelve la posición del puntero de escritura

## Clase fstream

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

```
1 #include <fstream>
2 #include <iostream>
3 using namespace std;
4 int main(){
5 fstream fich("fstream.dat", ios::in | ios::out
6 | ios::trunc | ios::binary);
7 fich << "abracadabra" << flush;
8 fich.seekg(OL, ios::end);
9 long lon = fich.tellg();
10 for(long i = OL; i < lon; ++i) {
11 fich.seekg(i, ios::beg);
12 if(fich.get() == 'a') {
13 fich.seekp(i, ios::beg);
14 fich << 'e';
15 }
16 }
17 cout << "Salida: ";
18 fich.seekg(OL, ios::beg);
19 for(long i = OL; i < lon; ++i)
20 cout << fich.get();
21 cout << endl;
22 fich.close();
23 }
```



Salida: uebrecedebre

## Contenido del tema

- 1 Introducción
  - Flujos de E/S
  - Clases y ficheros de cabecera
  - Flujos y búferes
- 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
- 6 Flujos asociados a ficheros
  - Introducción
  - Tipos de ficheros: texto y binarios
  - 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

## Funciones miembro

### `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

```



## Flujos asociados a strings

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.



Incluiríremos 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`

## 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
```