

INFORMÁTICA INDUSTRIAL I

Grado en Ingeniería en Tecnologías Industriales
3^{er} curso

7. Lenguaje C++: Manejo de Ficheros

Profesores: M. Abderrahim / F. Abu-Dakka / Á. Castro González
Departamento de Ingeniería de Sistemas y Automática

AGENDA

- *Streams*
- Crear un fichero de salida, abrir un fichero de entrada
- Ficheros Binarios
- Ficheros de Acceso Aleatorio
- Ficheros de Entrada y Salida
- Sobrecarga de los operadores << y >>
- Comprobar el estado de un *stream*

Streams

- En C++, los archivos se manejan con un tipo particular de *stream*.
- Un *stream* es una estructura de datos que se utiliza para manejar un “flujo de caracteres” y permitir poner o sacar de él tipos de datos estándar o clases definidas por el usuario.
- Usar *streams* facilita mucho el acceso a ficheros en disco, veremos que una vez que creamos un *stream* para un fichero, podremos trabajar con él igual que lo hacemos con *cin* o *cout*.

Crear un fichero de salida, abrir un fichero de entrada

- Crear un fichero mediante un objeto de la clase *ofstream*, y posteriormente lo leeremos mediante un objeto de la clase *ifstream*.
- Este sencillo ejemplo crea un fichero de texto y después visualiza su contenido en pantalla.

Ejemplo: Crear un fichero de salida, abrir un fichero de entrada

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

```
int main() {
    char cadena[128];
    // Crea un fichero de salida
    ofstream fs("nombre.txt");

    // Enviamos una cadena al fichero de salida:
    fs << "Hola, mundo" << endl;
    // Cerrar el fichero,
    // para luego poder abrirlo para lectura:
    fs.close();

    // Abre un fichero de entrada
    ifstream fe("nombre.txt");
```

```
// Leeremos mediante getline, si lo hiciéramos
// mediante el operador >> sólo leeríamos
// parte de la cadena:
    fe.getline(cadena, 128);

    cout << cadena << endl;

    return 0;
}
```

```
//A veces es necesario usar cin.sync() para sincronizar
//el buffer de entrada
```

```
WJRG-MacBookPro:Clase12 wladimir$ g++ fichero.cpp -o fichero
WJRG-MacBookPro:Clase12 wladimir$ ./fichero
Hola, mundo
```

Otro Ejemplo

- Veamos otro ejemplo sencillo, para ilustrar algunas *limitaciones* del operador `>>` para hacer lecturas, cuando no queremos perder caracteres.
- Supongamos que llamamos a este programa "*streams.cpp*", y que pretendemos que se autoimprima en pantalla.
- El resultado quizá no sea el esperado. El motivo es que el operador `>>` interpreta los espacios, tabuladores y retornos de línea como separadores, y los elimina de la cadena de entrada.

Otro Ejemplo

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

int main() {
    char cadena[128];
    ifstream fe("streams.cpp");

    while(!fe.eof()) {
        fe >> cadena;
        cout << cadena << endl;
    }
    fe.close();

    return 0;
}
```

Solución:

```
fe.getline(cadena,128);
```

```
C:\QtSDK\QtCreator\bin\qtcreator_process_stub.exe
#include
<iostream>
#include
<fstream>
using
namespace
std;
int
main()
{
    char
    cadena[128];
    ifstream
    fe("streams.cpp");

    while(!fe.eof()) {
        fe >> cadena;
        cout << cadena << endl;
    }
    fe.close();

    return 0;
}

Press <RETURN> to close this window...
```

```
C:\QtSDK\QtCreator\bin\qtcreator_process_stub.exe
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    char cadena[128];
    ifstream fe("main.cpp");

    while(!fe.eof()) {
        fe >> cadena;
        cout << cadena << endl;
    }
    fe.close();

    return 0;
}

Press <RETURN> to close this window...
```

Ejercicio

- Escribir una función (o funciones) para leer un fichero de texto línea a línea y separar las letras de los números:

- Esta es la salida por pantalla:

```
Las letras son: hola
Los numeros son: 1
Las letras son: d
Los numeros son: 2347
Las letras son: abc
Los numeros son: 34
Las letras son: deF
Los numeros son: 45
```

Fichero.txt

hola1

234d7

34abc

deF45

Pista: la función **bool isdigit(char ch)** nos dice si el parámetro es un número o no

Ficheros Binarios

- En general, usaremos **ficheros de texto** para almacenar información que pueda o deba ser manipulada con un editor de texto. Un ejemplo es un fichero fuente C++.
- Los **ficheros binarios** son más útiles para guardar información cuyos valores no estén limitados. Por ejemplo, para almacenar imágenes, o bases de datos.
- Un fichero binario permite almacenar estructuras completas, en las que se mezclen datos de cadenas con datos numéricos.

Ejemplo: Fiche

```
WJRG-MacBookPro:Clase12 wladimir$ g++ binario.cpp -o binario
WJRG-MacBookPro:Clase12 wladimir$ ./binario
Jose Luis
32
1.78
```

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

using namespace std;

struct tipoRegistro {
    char nombre[32];
    int edad;
    float altura;
};

int main() {
    tipoRegistro pepe;
    tipoRegistro pepe2;
    ofstream fsalida("prueba.dat " , ios::out |
ios::binary);
    strcpy(pepe.nombre, "Jose Luis");
    pepe.edad = 32;
    pepe.altura = 1.78;
```

```
    fsalida.write(reinterpret_cast <char *>(&pepe),
sizeof(tipoRegistro));
    //fsalida.write((char *)(&pepe), sizeof(tipoRegistro));

    fsalida.close();
    //-----
    ifstream fentrada("prueba.dat", ios::in | ios::binary);

    fentrada.read((char *)(&pepe2), sizeof(tipoRegistro));

    cout << pepe2.nombre << endl;
    cout << pepe2.edad << endl;
    cout << pepe2.altura << endl;

    fentrada.close();

    return 0;
```

También:

```
ifstream fentrada("prueba.dat", ios_base::in | ios_base::binary);
```

Ficheros Binarios

- Al declarar streams de las clases ofstream o ifstream y abrirlos en modo binario, tenemos que añadir los *flags ios::out* e *ios::in*, respectivamente, al *flag ios::binary*. Esto es necesario porque los valores por defecto para el modo son *ios::out* e *ios::in*, también respectivamente, pero al añadir el *flag ios::binary*, el valor por defecto no se tiene en cuenta.
- Cuando trabajemos con streams binarios usaremos las funciones *write* y *read*. En este caso nos permiten escribir y leer estructuras completas.
- En general, cuando usemos estas funciones necesitaremos hacer un casting, es recomendable usar el operador **reinterpret_cast**.

Ficheros de Acceso Aleatorio

- Hasta ahora sólo hemos trabajado con los ficheros secuencialmente, es decir, empezamos a leer o a escribir desde el principio, y avanzamos a medida que leemos o escribimos en ellos.
- Otra característica importante de los ficheros es la posibilidad de trabajar con ellos haciendo acceso aleatorio, es decir, poder hacer lecturas o escrituras en cualquier punto del fichero. Para eso disponemos de las funciones *seekp* y *seekg*, que permiten cambiar la posición del fichero en la que se hará la siguiente escritura o lectura. La 'p' es de *put* y la 'g' de *get*, es decir escritura y lectura, respectivamente.

Fichero de Acceso Aleatorio

- Otro par de funciones relacionadas con el acceso aleatorio son ***tellp*** y ***tellg***, que sirven para saber en qué posición del fichero nos encontramos.
- La función ***seekg*** nos permite acceder a cualquier punto del fichero, no tiene por qué ser exactamente al principio de un registro, la resolución de la funciones ***seek*** es de un byte.

Ficheros de Acceso Aleatorio

```
ostream& seekp ( streampos pos );
```

```
ostream& seekp ( streamoff off, ios_base::seekdir dir );
```

Value dir	offset is relative to...
ios_base::beg	beginning of the stream buffer
ios_base::cur	current position in the stream buffer
ios_base::end	end of the stream buffer

Ejemplo: Fichero Aleatorio

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

int main() {
    int i;
    char mes[][20] = {"Enero", "Febrero", "Marzo",
        "Abril", "Mayo", "Junio", "Julio",
        "Agosto", "Septiembre", "Octubre",
        "Noviembre", "Diciembre"};
    char cad[20];
    ofstream fsalida("meses.dat", ios::out | ios::binary);

    // Crear fichero con los nombres de los meses:
    cout << "Crear archivo de nombres de meses:" << endl;
    for(i = 0; i < 12; i++)
        fsalida.write(mes[i], 20);
    fsalida.close();
    ifstream fentrada("meses.dat", ios::in | ios::binary);

    // Acceso secuencial:
    cout << "\nAcceso secuencial:" << endl;
    fentrada.read(cad, 20);
```

```
do {
    cout << cad << endl;
    fentrada.read(cad, 20);
} while(!fentrada.eof());

fentrada.clear();
// Acceso aleatorio:
cout << "\nAcceso aleatorio:" << endl;
for(i = 11; i >= 0; i--) {
    fentrada.seekg(20*i, ios::beg);
    fentrada.read(cad, 20);
    cout << cad << endl;
}

// Calcular el número de elementos
// almacenados en un fichero:
// ir al final del fichero
fentrada.seekg(0, ios::end);
// leer la posición actual
pos = fentrada.tellg();
// El número de registros es el tamaño en
// bytes dividido entre el tamaño del
// registro:
cout << "\nNúmero de registros: " << pos/20
    << endl;
fentrada.close();
return 0;
}
```

Ejemplo: Fichero Aleatorio

```
WJRG-MacBookPro:Clase12 wladimir$ g++ aleatorio.cpp -o aleatorio
WJRG-MacBookPro:Clase12 wladimir$ ./aleatorio
Crear archivo de nombres de meses:

Acceso secuencial:
Enero
Febrero
Marzo
Abril
Mayo
Junio
Julio
Agosto
Septiembre
Octubre
Noviembre
Diciembre

Acceso aleatorio:
Diciembre
Noviembre
Octubre
Septiembre
Agosto
Julio
Junio
Mayo
Abril
Marzo
Febrero
Enero

Número de registros: 12
```

Ficheros de Entrada y Salida

- Ahora veremos cómo podemos trabajar con un stream simultáneamente en entrada y salida.
- Para eso usaremos la clase ***fstream***, que al ser derivada de ***ifstream*** y ***ofstream***, dispone de todas las funciones necesarias para realizar cualquier operación de entrada o salida.
- Hay que tener la precaución de usar la opción ***ios::trunc*** de modo que el fichero sea creado si no existe previamente.

Ejemplo: Ficheros de Entrada y Salida

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

int main() {
    char l;
    long i, lon;
    fstream fich("prueba.dat", ios::in | ios::out | ios::trunc | ios::binary);

    fich << "abracadabra" << flush;

    fich.seekg(0L, ios::end);
    lon = fich.tellg();
    for(i = 0L; i < lon; i++) {
        fich.seekg(i, ios::beg);
        fich.get(l);
        if(l == 'a') {
            fich.seekp(i, ios::beg);
            fich << 'e';
        }
    }
}
```

```
cout << "Salida:" << endl;
fich.seekg(0L, ios::beg);
for(i = 0L; i < lon; i++) {
    fich.get(l);
    cout << l;
}
cout << endl;
fich.close();

return 0;
}
```

```
WJRG-MacBookPro:Clase12 wladimir$ g++ entsal.cpp -o entsal
WJRG-MacBookPro:Clase12 wladimir$ ./entsal
Salida:
ebrecedebre
```

Sobrecarga de Operadores << y >>

- Una de las principales ventajas de trabajar con *streams* es que nos permiten sobrecargar los operadores << y >> para realizar salidas y entradas de nuestros propios tipos de datos.

Ejemplo: Sobrecarga Operador <<

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

class Registro {
public:
    Registro(const char *, int, const char *);
    const char* LeeNombre() const {
        return nombre;
    }
    int LeeEdad() const {
        return edad;
    }
    const char* LeeTelefono() const {
        return telefono;
    }

private:
    char nombre[64];
    int edad;
    char telefono[10];
};
```

```
Registro::Registro(const char *n, int e, const char *t) : edad(e) {
    strcpy(nombre, n);
    strcpy(telefono, t);
}

ostream& operator<<(ostream &os, Registro& reg) {
    os << "Nombre: " << reg.LeeNombre()
    << "\nEdad: " << reg.LeeEdad()
    << "\nTelefono: " << reg.LeeTelefono();

    return os;
}

int main() {
    Registro Pepe("José", 32, "61545552");

    cout << Pepe << endl;

    return 0;
}
```

```
WJRG-MacBookPro:Clase12 wladimir$ g++ sobrecarga.cpp -o sobrecarga
WJRG-MacBookPro:Clase12 wladimir$ ./sobrecarga
Nombre: José
Edad: 32
Telefono: 61545552
```

Comprobar Estado de un *stream*

- Hay varios *flags* de estado que podemos usar para comprobar el estado en que se encuentra un *stream*.
- Concretamente nos puede interesar si hemos alcanzado el fin de fichero, o si el *stream* con el que estamos trabajando está en un estado de error.
- La función principal para esto es ***good()***, de la clase *ios*.
- Después de ciertas operaciones con *streams*, a menudo no es mala idea comprobar el estado en que ha quedado el *stream*. Hay que tener en cuenta que ciertos estados de error impiden que se puedan seguir realizando operaciones de entrada y salida.

Comprobar Estado de un *stream*

- Otras funciones útiles son *fail()*, *eof()*, *bad()*, *rdstate()* o *clear()*.

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
goodbit	No errors (zero value iostate)	true	false	false	false	goodbit
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit
failbit	Logical error on i/o operation	false	false	true	false	failbit
badbit	Read/writing error on i/o operation	false	false	true	true	badbit

Fuente: <http://www.cplusplus.com/reference/ios/ios/fail/>

Comprobar Estado de un *stream*

- En el ejemplo de archivos de acceso aleatorio hemos usado *clear()* para eliminar el bit de estado *eofbit* del fichero de entrada, si no hacemos eso, las siguientes operaciones de lectura fallarían.
- Otra condición que conviene verificar es la existencia de un fichero.
- Cuando vayamos a leer un fichero que no podamos estar seguros de que existe, o que aunque exista pueda estar abierto por otro programa, debemos asegurarnos de que nuestro programa tiene acceso al *stream*.

Ejemplo: Comprobar si un Fichero Existe

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

int main() {
    char mes[20];
    ifstream fich("meses1.dat", ios::in | ios::binary);

    // El fichero meses1.dat no existe, este programa es una prueba de los bits de estado.
    if(fich.good()) {
        fich.read(mes, 20);
        cout << mes << endl;
    }
    else {
        cout << "Fichero no disponible" << endl;
        if(fich.fail()) cout << "Bit fail activo" << endl;
        if(fich.eof()) cout << "Bit eof activo" << endl;
        if(fich.bad()) cout << "Bit bad activo" << endl;
    }
    fich.close();
    return 0;
}
```

```
WJRG-MacBookPro:Clase12 wladimir$ g++ existe.cpp -o existe
WJRG-MacBookPro:Clase12 wladimir$ ./existe
Fichero no disponible
Bit fail activo
```

Ejemplo: Comprobar si un Fichero ya esta Abierto

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

int main() {
    char mes[20];
    ofstream fich1("meses.dat", ios::out | ios::binary);
    ifstream fich("meses.dat", ios::in | ios::binary);
    // El fichero meses.dat existe, pero este programa intenta abrir dos streams al mismo fichero, uno en
    // escritura y otro en lectura. Eso no es posible, se trata de una prueba de los bits de estado.
    fich.read(mes, 20);
    if(fich.good())
        cout << mes << endl;
    else {
        cout << "Error al leer de Fichero" << endl;
        if(fich.fail()) cout << "Bit fail activo" << endl;
        if(fich.eof()) cout << "Bit eof activo" << endl;
        if(fich.bad()) cout << "Bit bad activo" << endl;
    }
    fich.close();
    fich1.close();
    return 0;
}
```

```
WJRG-MacBookPro:Clase12 wladimir$ g++ abierto.cpp -o abierto
WJRG-MacBookPro:Clase12 wladimir$ ./abierto
Error al leer de Fichero
Bit fail activo
Bit eof activo
```