

Tema 3: Excepciones y Ficheros en C++

Excepciones en C++

C++ implementa manejo de excepciones: nos sirven para gestionar errores de una forma homogénea. Si un procedimiento no trata la excepción (error) producido *lanzar*á (o elevará) una excepción de modo que en el contexto superior se puede detectar una situación anormal.

El contexto superior puede tratar la excepción o ignorarla (en cuyo caso se pasa a su contexto superior).

Las palabras reservadas de C++ en relación con las excepciones son:

throw: declara que una función o método lanza una excepción.

También permite lanzar una excepción

try: declara un bloque dentro del cual se puede capturar una excepción.

catch: declara un bloque de tratamiento de excepción.

Un primer ejemplo: división por 0

<pre>#include <iostream> #include <exception> using namespace std; double divide(int dividendo, int divisor){ if (divisor == 0) throw exception(); //lanza excepcion y termina funcion return (double)dividendo/divisor; } int main(int argc, char *argv[]) { cout << "Division correcta: " << divide(1,2) << endl; /* Este programa termina automáticamente */ cout << "Division por cero: " << divide(1,0) << endl; system("PAUSE"); return EXIT_SUCCESS; }</pre>	<pre>#include <iostream> #include <exception> //hay que incluir esta libreria para el //manejo de excepciones VIP!!! using namespace std; double divide(int dividendo, int divisor){ if (divisor == 0) throw exception(); return (double)dividendo/divisor; } int main(int argc, char *argv[]) { try{ cout << "Division correcta: " << divide(1,2) << endl; /* Este programa captura y trata la excepción */ cout << "Division por cero: " << divide(1,0) << endl; cout << "Adios\n"; }catch(exception e){ //bloque tratamiento excepcion cout << "Ocurrio una excepcion: " << e.what() << endl; } system("PAUSE"); return EXIT_SUCCESS; }</pre>
Division correcta: 0.5 Runtime error	Division correcta: 0.5 Ocurrio una excepcion: St9exception

Cuando un bloque try se ejecuta sin lanzar excepciones, el flujo de control pasa a la instrucción siguiente al último bloque catch asociado con ese bloque try.

Dentro de un bloque try

```
try{
1---
2---
3---
4---
}catch( clase_excepcion1 e ) {
...
}
catch( clase_excepcion2 e ) {
...
}
cout << "seguir por aqui \n"
```

Si la excepción ocurre en la línea 2) el bloque try expira, las variables locales del try se destruyen y el flujo del programa pasa al bloque catch (puede haber varios) que procesa la excepción, y cuando se acaba no se retorna donde se produjo la excepción, sino que se sigue con la próxima sentencia que no es un catch(). En el ejemplo mostrado sería:

cout << "seguir por aqui \n"

Si ningún bloque catch lo procesa, el programa termina.

Por cada try debe haber al menos un catch, que entre paréntesis especifica un parámetro de excepción que representa el tipo de excepción que puede procesar ese catch

(Nota: C++ tiene definidas muchos tipos de excepciones)

Excepciones en C++

Cuando un método se encuentra una anomalía que no puede resolver, lo lógico es que lance (**throw**) una excepción, esperando que quien lo llamó directa o indirectamente la capture (**catch**) y maneje la anomalía. El propio método puede capturar y manejar dicha excepción.

Una vez lanzada (**throw**) una excepción, el sistema es responsable de encontrar a alguien que lo capture. Esos “alguien” es el conjunto de métodos en la pila de llamadas hasta que ocurrió el error.

Si ninguno de los métodos de la pila de llamadas es capaz de manejar la excepción el programa termina.

Supongamos que f1() llama a f2() y f2() llama a f3(). Si ocurre un error en f3() y no captura el error, bien porque no tenga try-catch asociado o bien porque teniéndolo ningún catch es del tipo de la excepción que ocurre → f3() pasa la excepción a f2() y si ésta no la procesa se la pasa a f1() y si ésta no la procesa se la pasa al main() y si esta no la procesa el programa termina.

Para capturar una excepción se coloca la porción de código que se desea controlar dentro de un bloque **try**. Si al ejecutarse el bloque try se produce una excepción el control pasa al bloque catch de tratamiento de la excepción.

Un primer ejemplo: división por 0

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

double divide( int dividendo, int divisor ){
    if ( divisor == 0 )
        throw exception(); //lanza una excepcion y termina
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    cout << "Division correcta: " << divide(1,2) << endl;
    /* Este programa termina automáticamente */
    cout << "Division por cero: " << divide(1,0) << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <iostream>
#include <exception> //hay que incluir esta libreria para el
                    //manejo de excepciones VIP!!!

using namespace std;

double divide( int dividendo, int divisor ){
    if ( divisor == 0 )
        throw exception();
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    try{
        cout << "Division correcta: " << divide(1,2) << endl;
        /* Este programa captura y trata la excepción */
        cout << "Division por cero: " << divide(1,0) << endl;
        cout << "Adios\n";
    }catch( exception e ){
        cout << "Ocurrio una excepcion: " << e.what() << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Cuando un bloque try se ejecuta sin lanzar excepciones, el flujo de control pasa a la instrucción siguiente al último bloque catch asociado con ese bloque try.

Pantalla:

```
Division correcta: 0.5
Runtime error
```

Si eliminamos **if () throw**

Pantalla:

```
Division correcta: 0.5
Division por cero: 1.#INF
Presione una tecla para continuar . . .
```

Pantalla:

```
Division correcta: 0.5
Ocurrio una excepcion: St9exception
Presione una tecla para continuar . . .
```

Excepciones en C++

Además de usar las diferentes tipos de excepciones que incorpora C++, nosotros podemos definir nuestras propias excepciones. Las excepciones se definen como clases:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class ExcepcionDivCero{
public:
    ExcepcionDivCero() { //constructor
        char m[] = "Excepcion: division por 0";
        mensaje=new char[strlen(m)+1];
        strcpy(mensaje, m);
    }
    const char *what() const{ return mensaje; }
private:
    char *mensaje;
};

double divide( int dividendo, int divisor ){
    if ( divisor == 0 )
        throw ExcepcionDivCero();
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    cout << "Division correcta:" << divide(1,2) << endl;
    /* Este programa sale automáticamente...
       ya que no captura la excepción...*/
    cout << "Division por cero:" << divide(1,0) << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

El programa termina aquí. Esto no se ejecuta

```
#include <cstdlib>
#include <iostream>

using namespace std;

class ExcepcionDivCero{
public:
    ExcepcionDivCero( ): mensaje( "Excepcion: division por 0" ) {}
    const char *what() const{ return mensaje; }
private:
    char *mensaje;
};

double divide( int dividendo, int divisor ){
    if ( divisor == 0 )
        throw ExcepcionDivCero(); //lanza excepción de tipo ExcepcionDivCero
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    cout << "Division correcta:" << divide(1,2) << endl;
    /* Este programa captura y trata la excepción */
    try{
        cout << "Division por cero:" << divide(1,0) << endl;
        cout << "Adios\n";
    }catch( ExcepcionDivCero e ){
        cout << "Ocurrio una excepcion: " << e.what() << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Se produce la excepción y se captura por estar en un bloque try. El bloque catch correspondiente lo trata

Sobrecargamos el método `what()` para mostrar un mensaje personalizado.

Pantalla:

```
Division correcta: 0.5
Runtime error
```

Si eliminamos `if () throw`

Pantalla:

```
Division correcta: 0.5
Division por cero: 1.#INF
Presione una tecla para continuar . . .
```

Pantalla:

```
Division correcta: 0.5
Ocurrio una excepcion: Excepcion: division por 0
Presione una tecla para continuar . . .
```

En este ejemplo, la excepción la hemos creado a partir de cero.

Es más lógico crear la excepción (mediante herencia) a partir de la clase `exception` o de cualquiera de las excepciones que incorpora C++ para aprovechar su funcionalidad.

Excepciones en C++

Las clases de excepciones pueden heredar de otras clases de excepción o no heredar de nadie. La declaración de la clase básica `exception` está en la cabecera `<exception>`. Aunque no es necesario, se suelen hacer subclases de `exception` o de otras clases de excepción que hayamos definido. Esto se hace porque en los bloques `catch` se capturan las excepciones de la clase que aparece o de cualquier subclase de ella. La herencia debe ser pública.

throw() Indica que el método no lanza excepciones.

Cuando heredo de `exception` debo sobrecargar el método `what` exactamente con esa cabecera.

El `&` de `catch(exception &e)` es para que tenga comportamiento polimórfico, de forma que si se lanza una excepción que hereda de `exception` lo trate.

Pantalla:

Division correcta: 0.5
Ocurrió una excepción: Division por 0
Presione una tecla para continuar . . .

Si quito la línea verde:

Division correcta: 0.5
Ocurrió una excepción: Valor no valido
Presione una tecla para continuar . . .

Si quito el `&` `catch(exception e)` no hay polimorfismo y se ejecuta el `what()` de `exception` y no el de `Error`
`exception` atrapa `Error` ya que deriva de `exception`

Pantalla:

Division correcta: 0.5
Ocurrió una excepción: St9exception
Presione una tecla para continuar . . .

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

//creamos una clase Error (hija de exception) con un mensaje por defecto
class Error: public exception {
public:
    Error(char *m="Valor no valido"): exception(), mensaje(m) { }
    const char *what() const throw() {
        return mensaje;
    }
private:
    char *mensaje;
};

double divide( int dividendo, int divisor ){
    if ( divisor == 0 ) throw Error("Division por 0");
    return (double)dividendo/divisor;
}

double raiz(double d) {
    if (d<0) throw Error(); //lo crea con mensaje por defecto
    return sqrt(d);
}

int main(int argc, char *argv[]) {
    cout << "Division correcta:" << divide(1,2) << endl;
    try{ /* Este programa captura y trata la excepción */
        cout << "Division por cero:" << divide(1,0) << endl;
        cout << "Raiz negativa:" << raiz(-5) << endl;
    } catch( exception &e ){
        cout << "Ocurrió una excepción: " << e.what() << endl;
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Al poner en el `catch` `exception`, captura las excepciones de objetos `exception` y de cualquier clase que derive de `exception` → Como `Error` deriva de `Exception` también lo captura.

Al tener el `&` (es decir, al ser una referencia), y ser el método `what()` de `exception` virtual, el método que se ejecuta sería el del tipo al que apunta la referencia (no el del tipo de la referencia): como la referencia apunta a un objeto de tipo `Error` → se ejecuta el `what()` de `Error`.

Además con el `&` aunque no hay clases derivadas es más eficiente ya que al ser una referencia evitamos que se pase por valor y nos ahorramos una copia → usar siempre `&`

Si no tuviera el `&` la excepción `Error` la capturaría el `catch(exception e)` ya que deriva de `exception` pero no tendría comportamiento polimórfico y el método `what()` que se ejecutaría sería el del `exception` en lugar del de `Error`

Excepciones en C++

Las clases de excepciones pueden heredar de otras clases de excepción o no heredar de nadie. La declaración de la clase básica `exception` está en la cabecera `<exception>`. Aunque no es necesario, se suelen hacer subclases de `exception` o de otras clases de excepción que hayamos definido. Esto se hace porque en los bloques `catch` se capturan las excepciones de la clase que aparece o de cualquier subclase de ella. La herencia debe ser pública.

throw() Indica que el método no lanza excepciones.

Cuando heredo de `exception` debo sobrecargar el método `what` exactamente con esa cabecera.

El `&` de `catch(exception &e)` es para que tenga comportamiento polimórfico, de forma que si se lanza una excepción que hereda de `exception` lo trate.

Pantalla:

Division correcta: 0.5
Ocurrió una excepcion: `Division por cero`
Presione una tecla para continuar . . .

Si quito el `&` `catch(exception e)` no hay polimorfismo y se ejecuta el `what()` de `exception` y no el de `ExcepcionDivCero2`
`exception` atrapa `ExcepcionDivCero2` ya que deriva de `exception`

Pantalla:

Division correcta: 0.5
Ocurrió una excepcion: `St9exception`
Presione una tecla para continuar . . .

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

class ExcepcionDivCero: public exception{
public:
    ExcepcionDivCero( ) : mensaje( "Excepcion: division por 0" ) {}
    const char *what() const throw() {
        return mensaje;
    }
private:
    char *mensaje;
};

class ExcepcionDivCero2: public exception{
public:
    ExcepcionDivCero2( ) /*: exception( )*/ {}
    const char *what() const throw() {
        return "Division por cero"; //paso el mensaje que quiero mostrar
    }
};

double divide( int dividendo, int divisor ){
    if ( divisor == 0 )
        throw ExcepcionDivCero2( );
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    cout << "Division correcta:" << divide(1,2) << endl;
    /* Este programa captura y trata la excepción */
    try{
        cout << "Division por cero:" << divide(1,0) << endl;
    } catch( exception &e ){
        cout << "Ocurrió una excepcion: " << e.what() << endl;
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

En realidad `exception` tiene internamente su propio atributo con el mensaje.

El `char *mensaje` de `ExcepcionDivCero` es distinto del propio que tiene `exception`

No es necesario crear una clase con un atributo `mensaje` como se muestra en `ExcepcionDivCero2`

Interesa si queremos indicar qué mensaje queremos que se muestre como ocurre en el ejemplo de la clase `Error` de la transparencia anterior

Excepciones en C++

Como es posible que nuestras excepciones no deriven de ninguna clase base, C++ permite una sentencia `catch` que capture *cualquier clase que se eleve*. La forma de hacer esto es con `catch (...)`

Un bloque `try` puede tener varios bloques `catch` asociados. Cada bloque `catch` es un ámbito diferente.

Un bloque `try` puede tener varios bloques `catch`, tantos como excepciones diferentes hay que manejar.

Cada `catch` tendrá un parámetro `exception`, de alguna clase derivada de ésta o una clase definida por el usuario (no tiene por que derivar de `exception`)

Cuando se lanza una excepción, los bloques `catch` se van probando en el orden indicado, de forma que el orden de los `catch` debe ser tal, que ninguno de ellos englobe al resto. Por ejemplo, si el primer bloque `catch` es de tipo `exception`, ningún otro bloque que le siga con un parámetro de alguna de sus derivadas podría alcanzarse.

En el ejemplo si 2) va al principio nunca se ejecutaría 1) ya que 1) deriva de 2).

Pantalla:

Division correcta: 0.5
Ocurrio una excepcion: Division por 0

Si quito la línea verde:

Division correcta: 0.5
Ocurrio una excepcion: Valor no valido

Si ocurre un excepción int:

Division correcta: 0.5
Ocurrio una excepción que no hereda de

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

//creamos una clase EValorNoValido con un mensaje por defecto
class EValorNoValido: public exception {
public:
    EValorNoValido(char *m="Valor no valido"): mensaje( m ) { }
    const char *what() const throw(){
        return mensaje;
    }
private:
    char *mensaje;
};

//supongamos que solo podemos dividir por valores positivos...
double divide( int dividendo, int divisor ){
    if ( divisor == 0 ) throw EValorNoValido("Division por 0");
    else if ( divisor < 0 ) throw EValorNoValido( );
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    cout << "Division correcta:" << divide(1,2) << endl;
    /* Este programa captura y trata la excepción */
    try{
        ... //aquí pueden ocurrir otras excepciones
        cout << "Division por cero:" << divide(1,0) << endl;
        cout << "Division negativa:" << divide(1,-5) << endl;
    }catch(EValorNoValido &e ){ //1
        cout << "Ocurrio una excepcion: " << e.what() << endl;
    }catch( exception &e ){ //2
        cout << "Ocurrio una excepcion: " << e.what() << endl;
    }catch( ... ){ //3
        cout << "Ocurrio una excepcion que no hereda de exception\n";
    }
    system("PAUSE"); //4
    return EXIT_SUCCESS;
}
```

Si en el bloque `try` ocurriera una excepción distinta de `EValorNoValido` lo capturaría el 2) que captura excepciones de tipo `exception` o de cualquiera de sus clases derivadas y con el `&` tendría comportamiento polimórfico ejecutándose el `what()` de la clase a la que apunta la referencia.

El 3) es por si ocurre una excepción que no hereda de `exception`

Una vez ocurra una excepción en el `try`, el resto de líneas del `try` no se ejecutan, las variables locales del `try` se destruyen y el flujo del programa pasa al bloque `catch` que capture la excepción (se ejecuta aquel `catch` cuyo tipo de parámetro asociado coincida con el tipo de excepción que ocurre o derive de éste). Una vez ejecutado un bloque `catch` el resto de bloques `catch` se ignoran y el programa continua por las líneas que sigan después de los `catch` que hay (en el ejemplo sería //4)

Si no existe ninguno que coincida la función donde esta el bloque `try` termina y el programa intenta buscar un bloque `try` en la función que hizo la llamada y así sucesivamente.

Si no existe ninguno que lo capture el programa termina.

Excepciones en C++

Se pueden tener varios **bloques try anidados** de modo que, si no se encuentran manejadores de excepción en un bloque, se pasa a buscarlos al bloque inmediatamente superior. Si se sale de todos los bloques anidados sin encontrar un manejador, el programa terminará (por defecto).

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

//creamos una clase EValorNoValido con un mensaje por defecto
class EValorNoValido: public exception {
public:
    EValorNoValido(char *m="Valor no valido"): mensaje( m ) { }
    const char *what() const throw(){
        return mensaje;
    }
private:
    char *mensaje;
};

double divide( int dividendo, int divisor ){
    if ( divisor == 0 ) throw EValorNoValido("Division por 0");
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    cout << "Division correcta:" << divide(1,2) << endl;
    /* Este programa captura y trata la excepción */
    try{
        /* código susceptible de elevar excepciones */

        try{

            /* Este bloque try lanza una excepcion exception() que
            no maneja su correspondiente catch*/
            throw exception();

        }catch(EValorNoValido &e){
            cout << "Ocurrio una excepcion: " << e.what() << endl;
        }

    }catch( exception e ){
        cout << "Ocurrio una excepcion: " << e.what() << endl;
    }catch( ... ){
        cout << "Ocurrio una excepcion que no hereda de exception\n";
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

La excepción `exception()` no es capturada por el `catch (EValorNoValido &e)`, pero lo captura el `catch (exception e)` del try superior.

Excepciones en C++

Tenga en cuenta que **el bloque try define un ámbito** (ya que lleva { } y todo lo que declare dentro de las llaves es local) y los objetos declarados en ese ámbito no están disponibles en el ámbito del bloque catch que, en su caso, se ejecute.

Es posible **relanzar la misma excepción** que se está tratando. Esto se hará en casos en que el tratamiento de la excepción no se haga completamente en un único manejador. Si utilizamos un bloque catch que da nombre a la excepción capturada, podemos volver a lanzarla con `throw <nombre>;`

En el caso de estar en un manejador `catch(...)`, se puede relanzar la excepción con `throw;`

Todas las variables declaradas dentro de un bloque { } son locales a dicho bloque. Por tanto toda variable declarada dentro de un bloque try { } son locales al try.

De la misma forma, todas las variables declaradas dentro de un bloque catch { } son locales al catch.

Si `char *array` lo hubiera declarado dentro del try, solo lo podría usar en el try.

Como lo quiero usar tanto en el try como en el catch lo he declarado fuera de ambos, para poder usarlo en ambos.

En el ejemplo:

- Se ha ejecutado `catch(...)`
- Se ha ejecutado `delete [] array;`
`throw;`
- Se ha ejecutado `catch(exception e)`

Pantalla

Borrando...
Ocurrió una excepción: St9exception
Presione una tecla para continuar . . .

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

//creamos una clase EValorNoValido con un mensaje por defecto
class EValorNoValido: public exception {
public:
    EValorNoValido(char *m="Valor no valido"): mensaje( m ) { }
    const char *what() const throw(){
        return mensaje;
    }
private:
    char *mensaje;
};

double divide( int dividendo, int divisor ){
    if ( divisor == 0 ) throw EValorNoValido("Division por 0");
    return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
    char *array;
    /* Este programa captura y trata la excepción */
    try{
        /* bloque susceptible de elevar excepciones */
        try{
            /* reserva de memoria */
            array = new char[200];
            /* Este bloque try lanza una excepcion que
            maneja su correspondiente catch pero no completamente*/
            throw exception();
        }catch( ... ){
            /* Queremos liberar la memoria, aunque la excepción
            se tratará en otro bloque try, si procede... */
            delete [] array;
            cout << "Borrando... \n";
            throw; //relanzo la excepcion (aunque la he tratado en parte
            //aqui) y la captura el catch (exception e) del try externo
        }catch( exception &e ){
            cout << "Ocurrió una excepción: " << e.what() << endl;
        }catch( ... ){
            cout << "Ocurrió una excepcion que no hereda de exception\n";
        }
        system("PAUSE");
        return EXIT_SUCCESS;
    }
```


Excepciones en C++

El comportamiento por defecto cuando no se encuentra un manejador de excepción es la terminación del programa.

Este comportamiento se puede modificar con la función **set_terminate(void(*)())**.

void(*) () → puntero a una función void

Es buena idea definir en el programa una función void que determine el comportamiento del programa cuando termina de forma anormal

Podemos tener varias funciones void y a lo largo del programa indicar cual queremos que se ejecute.

Pantalla

Comienzo del programa
Final anormal del programa!!
Presione una tecla para continuar . . .

Si comentamos lo gris, lo que saldría por pantalla sería lo siguiente:

Pantalla

Comienzo del programa
Error del programa!!
Presione una tecla para continuar . . .

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

double divide( int dividendo, int divisor ){
    if ( divisor == 0 )
        throw exception();
    return (double)dividendo/divisor;
}

void terminar( ) { //funcion void
    cout << "Final anormal del programa!!\n";
    system("PAUSE");
    exit(EXIT_SUCCESS);
}

//es bueno tener una funcion de este tipo... podemos tener varias
void terminar2( ) { //funcion void
    cout << "Error del programa!!\n";
    system("PAUSE");
    exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
    /* Este programa no captura la excepción */
    cout << "Comienzo del programa\n";
    set_terminate( terminar ); /*de aqui para abajo, si ocurre una
    excepcion que no se captura el programa termina ejecutando la funcion
    terminar*/
    divide(1,0); //pruebe a comentar esta linea
    set_terminate( terminar2 ); /*de aqui para abajo, si ocurre una
    excepcion que no se captura el programa termina ejecutando la funcion
    terminar*/
    divide(1,0);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Excepciones en C++

Se puede especificar una lista de excepciones que una función o método puede lanzar. Para ello, se escribe una lista **throw([lista clases excepcion])** después del nombre y lista de parámetros de la función en cuestión. A la hora de heredar, no es posible sobrescribir un método dándole menores restricciones en cuanto a lanzamiento de excepciones, esto es, un método que sobrescribe a otro puede lanzar, a lo sumo, las mismas excepciones que el método sobrescrito.

No es obligatorio indicarlo, pero si se indica entonces la función o método sólo puede lanzar excepciones de los tipos indicados (o derivados de los indicados). Si la función lanza una excepción de un tipo distinto a los indicados el programa falla en tiempo de ejecución.

Si no se indica una lista throw, la función puede lanzar excepciones de cualquier tipo.

Con **throw()** se declara que una función o método no lanza excepciones.
Si dentro de what() lanzo una excepción en tiempo de compilación no da error pero si lo dará en tiempo de ejecución

El compilador no da error si no hay un throw para cada una de las excepciones indicadas en la lista. Tampoco da error si pongo un throw con una excepción que no está en la lista.
El error, de ocurrir, será en tiempo de ejecución.

En el programa indicamos que divide solo puede lanzar excepciones de tipo exception y EValorNoValido o que deriven de ellas.
Si lanza otras distintas → error en tiempo de ejecución

Pantalla

Ocurrió una excepcion: St9exception
Presione una tecla para continuar . . .

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

//creamos una clase EValorNoValido con un mensaje por defecto
class EValorNoValido: public exception {
public:
    EValorNoValido(char *m="Valor no valido"): mensaje( m ) { }
    const char *what() const throw(){
        return mensaje;
    }
private:
    char *mensaje;
};

double divide( int diviendo, int divisor )
    throw (exception, EValorNoValido)
{
    if ( divisor == 0 ) throw EValorNoValido("Division por 0");
    return (double)diviendo/divisor;
}

int main(int argc, char *argv[])
{
    char *array;
    /* Este programa captura y trata la excepción */
    try{
        /* bloque susceptible de elevar excepciones */
        try{
            /* reserva de memoria */
            array = new char[200];
            /* Este bloque try lanza una excepcion que
            no manejan sus correspondientes catch*/
            throw exception();
        }catch( ... ){
            /* Queremos liberar la memoria, aunque la excepción
            se tratará en otro bloque try, si procede... */
            delete [] array;
            throw;
        }
    }catch( exception e ){
        cout << "Ocurrió una excepcion: " << e.what() << endl;
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Excepciones en C++

Siempre es interesante a la hora de definir una función indicar si va a lanzar una excepción. Además esta lista `throw` pasa a formar parte del prototipo de la función. De esta forma, el que utiliza la función sabe que tiene que tratar las excepciones indicadas en la lista.

Si ponemos una lista `throw()` vacía indica que la función no puede lanzar ninguna excepción.

Si no se indica una lista `throw`, la función puede lanzar excepciones de cualquier tipo.

`float f1() throw ();` //f1() no puede lanzar ninguna excepción

`float f2();` //f2() puede lanzar cualquier excepción

`float f3() throw(int, char, exception);` //f3() puede lanzar excepciones de tipo int, char y exception (o derivadas de ellas)

`float f3() { };` //error en la implementación, falta indicar la lista `throw`

`float f3() throw(int, char, exception) { //poner aquí el código que sea };` //correcto

Con `throw()` se declara que una función o método no lanza excepciones.

Si dentro de `what()` lanzo una excepción en tiempo de compilación no da error pero si lo dará en tiempo de ejecución

El compilador no da error si no hay un `throw` para cada una de las excepciones indicadas en la lista. Tampoco da error si pongo un `throw` con una excepción que no está en la lista.

El error, de ocurrir, será en tiempo de ejecución.

En el programa indicamos que divide solo puede lanzar excepciones de tipo `exception` y `EValorNoValido` o que deriven de ellas.

Si lanza otras distintas → error en tiempo de ejecución

Pantalla

Final anormal del programa!!
Presione una tecla para continuar . . .

Si divide no tuviera

`throw(exception, EValorNoValido)`

Ocurrió una excepción que no hereda de `exception`

Presione una tecla para continuar . . .

Si `divide(1,0)` no estuviera:

Ocurrió una excepción tipo cadena

Presione una tecla para continuar . . .

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

```
//creamos una clase EValorNoValido con un mensaje por defecto
class EValorNoValido: public exception {
public:
```

```
    EValorNoValido(char *m="Valor no valido"): mensaje( m ) { }
```

```
    const char *what() const throw(){
```

```
        return mensaje;
```

```
    }
```

```
private:
```

```
    char *mensaje;
```

```
};
```

```
void terminar() { //funcion void
```

```
    cout << "Final anormal del programa!!\n";
```

```
    system("PAUSE"); exit(EXIT_SUCCESS);
```

```
}
```

```
double divide( int dividendo, int divisor )
```

```
    throw(exception, EValorNoValido) //solo puede lanzar estos
```

```
{ //2 tipos de excepciones
```

```
    if ( divisor == 0 )
```

```
        throw 1; //excepcion de tipo int!!!!
```

```
    return (double)dividendo/divisor;
```

```
}
```

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

```
{
```

```
    set_terminate( terminar ); //si aborta se ejecuta terminar( )
```

```
    /* Este programa captura y trata la excepción */
```

```
    try{
```

```
        /* bloque susceptible de elevar excepciones */
```

```
        divide(1,0);
```

```
        throw "excepcion"; //lanza excepción de tipo const char *
```

```
    } catch( exception &e ){
```

```
        cout << "Ocurrió una excepción: " << e.what() << endl;
```

```
    } catch( const char * ){
```

```
        cout << "Ocurrió una excepción tipo cadena\n";
```

```
    } catch( ... ){
```

```
        cout << "Ocurrió una excepción que no hereda de exception\n";
```

```
    }
```

```
    system("PAUSE");
```

```
    return EXIT_SUCCESS;
```

```
}
```

Ejemplo: Programa que calcula la solución de una ecuación de 2º grado

Pantalla

Las raices son:-1 y -1
Ocurrio una excepcion: Radicando
negativo
Presione una tecla para continuar . . .

```
#include <cstdlib>
#include <iostream>
#include <exception>
#include <math.h>
using namespace std;

//creamos una clase EValorNoValido con un mensaje por defecto
class EValorNoValido: public exception {
public:
    EValorNoValido(char *m="Valor no valido"): mensaje( m ) { }
    const char *what() const throw(){
        return mensaje;
    }
private:
    char *mensaje;
};

void terminar( ) { //funcion void
    cout << "Final anormal del programa!!\n";
    system("PAUSE");  exit(EXIT_SUCCESS);
}

void raices(int a, int b, int c) throw (exception, EValorNoValido) {
    double disc, r1, r2;
    if (b*b<4*a*c) throw EValorNoValido("Radicando negativo");
    if(a==0) throw EValorNoValido("a igual a 0");
    disc=sqrt(b*b-4*a*c);
    r1 = (-b-disc)/(2*a);
    r2 = (-b+disc)/(2*a);
    cout << "Las raices son:" << r1 <<" y " << r2 << endl;
}

int main(int argc, char *argv[]) {
    set_terminate( terminar ); //si aborta se ejecuta terminar( )
    /* Este programa captura y trata la excepción */
    try {
        raices(1, 2, 1); // dentro de raices() se lanza la excepción
        raices(2, 1, 2);
    } catch ( exception &e ){
        cout << "Ocurrio una excepcion: " << e.what() << endl;
    } catch( ... ){
        cout << "Ocurrio una excepcion que no hereda de exception\n";
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Ejercicio: Dado el siguiente código indica la salida que se genera por pantalla

```
#include <cstdlib>
#include <iostream>
#include <exception>
#include <math.h>
using namespace std;

class ENoValido;
void divide(int a, int b);
void raices(int a, int b, int c) throw (exception, ENoValido);
void terminar( );
void proceso(int valor);

class ENoValido: public exception {
    const char *mensaje;
public:
    ENoValido(const char *m="Valor no valido")
        : mensaje( m ) { }
    const char *what() const throw(){
        return mensaje;
    }
};

void terminar( ) { //funcion void
    cout << "Final anormal del programa!!\n";
    system("PAUSE");
    exit(EXIT_SUCCESS);
}

int main() {
    set_terminate( terminar );
    try{
        proceso(0);
    }catch( ... ){
        cout << "Algo raro ha ocurrido\n";
    }

    cout << "0000\n";

    try{
        proceso(3);
    }catch(ENoValido &e ){
        cout << "Error: " << e.what() << endl;
    }catch( exception &e ){
        cout << "Error: " << e.what() << endl;
    }catch( ... ){
        cout << "Error no identificado\n";
    }

    cout << "Continuacion del programa\n";
    divide(1,0);
    cout << "Fin correcto del programa\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}

void divide(int a, int b) {
    if ( a == 0 && b == 0 )
        throw ENoValido();
    else if ( b == 0 )
        throw exception();
    cout << a << "/" << b << " = " << a/b << endl;
}

void raices(int a, int b, int c) throw (exception, ENoValido) {
    double disc, r1, r2;
    if (b*b<4*a*c)
        throw ENoValido("Radicando negativo");
    if(a==0)
        throw ENoValido("a igual a 0");
    disc=sqrt(b*b-4*a*c);
    r1 = (-b-disc)/(2*a);
    r2 = (-b+disc)/(2*a);
    cout << "Raices: " << r1 << " y " << r2 << endl;
}

void proceso(int valor) {
    do {
        valor++;
        try {
            switch(valor) {
                case 1: raices(1, 2, 1);
                    cout << "1111\n";
                case 2: raices(2, 1, 2);
                    cout << "2222\n";
                    break;
                case 3: raices(0, 1, 2);
                    cout << "3333\n";
                    break;
                case 4: divide(10,2);
                    divide( 1,2);
                    divide(10,0);
                    cout << "4444\n";
                    break;
                case 5: throw int();
                    cout << "5555\n";
                case 6: divide(8,2);
                    divide(0,0);
                    cout << "6666\n";
                    break;
                default:cout << "7777\n";
            }
        }catch ( exception &e ){
            cout << "Error: " << e.what() << endl;
            if (valor < 4)
                throw;
        } catch( ... ){
            cout << "Error desconocido\n";
        }
    } while (valor<10);
}
```

SOLUCION:

```
Raices: -1 y -1
1111
Error: Radicando negativo
Algo raro ha ocurrido
0000
10/2 = 5
1/2 = 0
Error: std::exception
Error desconocido
8/2 = 4
```

```
Error: Valor no valido
7777
7777
7777
7777
Continuacion del programa
Final anormal del programa!!
Presione una tecla para continuar . . .
```

String

Cadenas con <string>

C++, en su biblioteca estándar, nos provee de la clase **string**, que resuelve muchos problemas clásicos con las cadenas C.

La clase string permite trabajar con cadenas de forma cómoda: no es necesario indicar la longitud de la cadena como ocurre en C, sino que esta se asigna automáticamente al leer mediante cin >>, tampoco hay que preocuparse por reservar o liberar memoria o por que se desborde la cadena.

La clase string proporciona métodos para comparar cadenas, buscar y extraer subcadenas, copiar, concatenar, etc.

Puede buscar una referencia completa en Internet.

Por ejemplo:

<http://www.msoe.edu/eecs/cese/resources/stl/string.htm>

<http://www.cppreference.com/cppstring/>

La cabecera está en <string>. ← necesario para usar string

Existen varios constructores y se definen operadores como la concatenación (+, +=) y las comparaciones (<, >, ==, !=).

Se pueden transformar en una cadena C normal (método c_str)

Versión C++

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
    string c1, c2;
    cin >> c1; //pide una cadena por teclado
    c2=c1; //copia c1 en c2
    string cad1 = "Una cadena";
    string *cad2 = new string("Otra cadena");
    cad1 += " y " + *cad2 + "\n"; //concatena

    cout << cad1;

    delete cad2;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Se puede sustituir por:

```
sprintf( tmp, "%s y %s\n" cad1, cad2 );
```

Versión C

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *cad1="Una cadena";
    char *cad2;
    char *tmp;
    cad2 = strdup( "Otra cadena" );

    /* Hay que reservar espacio suficiente!! */
    tmp = (char*)malloc(
        (strlen(cad1)+
        strlen(" y ")+
        strlen(cad2)+
        strlen("\n")+1)*sizeof(char) );

    strcpy( tmp, cad1 );
    strcat( tmp, " y " );
    strcat( tmp, cad2 );
    strcat( tmp, "\n" );

    printf( "%s", tmp );

    free(tmp);
    free(cad2);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Utilizar el tipo string hace que las aplicaciones sean más fáciles de programar:

Versión C (char *)	Versión C++ (string)
<pre> #include <iostream> //cin, cout using namespace std; class Persona { char *nombre; const int edad; //la edad no se puede modificar public: Persona(const char *nom, int e); Persona(const Persona& p); virtual ~Persona(); Persona& operator=(const Persona& p); int getEdad() const { return this->edad; } const char *getNombre() const { return this->nombre; } void setNombre(const char *nom); bool operator==(Persona p) const; }; Persona::Persona(const char *nom, int e): edad(e) { this->nombre=new char[strlen(nom)+1]; strcpy(this->nombre, nom); //this->edad=e; //ERROR edad es constante } Persona::~Persona() { delete [] this->nombre; } Persona::Persona(const Persona& p): edad(p.edad) { this->nombre=new char[strlen(p.nombre)+1]; strcpy(this->nombre, p.nombre); } //necesario porque edad no puede cambiar Persona& Persona::operator=(const Persona& p) { if (this != &p) { delete [] this->nombre; this->nombre=new char[strlen(p.nombre)+1]; strcpy(this->nombre, p.nombre); } return *this; } void Persona::setNombre(const char *nombre) { delete [] this->nombre; this->nombre=new char[strlen(nombre)+1]; strcpy(this->nombre, nombre); } bool Persona::operator==(Persona p) const { return (this->edad==p.edad && strcmp(this->nombre, p.nombre)==0); } ostream& operator<<(ostream &s, const Persona &p) { s << p.getNombre() << " (" << p.getEdad() << ")"; return s; } </pre>	<pre> #include <iostream> //cin, cout #include <string> //string using namespace std; class Persona { string nombre; const int edad; //la edad no se puede modificar public: Persona(string nom, int e); //Persona(const Persona& p); //NO NECESARIO //virtual ~Persona(); //NO NECESARIO Persona& operator=(const Persona& p); int getEdad() const { return this->edad; } string getNombre() const { return this->nombre; } void setNombre(string nom); bool operator==(Persona p) const; }; Persona::Persona(string nom, int e): edad(e) { this->nombre=nom; //this->edad=e; //ERROR edad es constante } /* Persona::~Persona() { //NO HAY QUE HACER NADA } Persona::Persona(const Persona& p): edad(p.edad) { this->nombre=p.nombre; } */ //necesario porque edad no puede cambiar Persona& Persona::operator=(const Persona& p) { if (this != &p) { this->nombre=p.nombre; } return *this; } void Persona::setNombre(string nombre) { this->nombre=nombre; } bool Persona::operator==(Persona p) const { return (this->edad==p.edad && strcmp(this->nombre, p.nombre)==0); } ostream& operator<<(ostream &s, const Persona &p) { s << p.getNombre() << " (" << p.getEdad() << ")"; return s; } </pre>

Nota: Si edad no fuera constante no sería necesario implementar el operator= al utilizar string (ya que no hay punteros)

Cadenas con <string>

Las cadenas de C++ sobrecargan también el operador [] de modo que se pueden usar como arrays.

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
    string strCPP = "Una cadena C++\n";
    char *strC = "Una cadena C\n";
    int tamCPP, tamC;

    tamCPP = strCPP.size();
    tamC = strlen( strC );

    for ( int i=0; i<tamCPP; i++ )
        cout << " " << strCPP[i];
    for ( int i=0; i<tamC; i++ )
        cout << " " << strC[i];

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

void funcion( string str ){
    for ( int i=0; i<str.size(); i++ )
        str[i] = '.';
}

int main(int argc, char *argv[])
{
    string str = "ORIGINAL";
    funcion(str);
    cout << str;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Atención: las cadenas C++ (string), al contrario que los arrays, son objetos, no punteros, de modo que se pasan por copia cuando son argumentos de funciones:

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

void f( string str ){
    for ( int i=0; i<str.size(); i++ )
        str[i] = '.';
}

int main(int argc, char *argv[])
{
    string str = "ORIGINAL";
    f( str );
    cout << str;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Paso por
referencia

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

void f( string &str ){
    for ( int i=0; i<str.size(); i++ )
        str[i] = '.';
}

int main(int argc, char *argv[])
{
    string str = "ORIGINAL";
    f( str );
    cout << str;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Cadenas con <string>

String constructors	create strings from arrays of characters and other strings
String operators	+ concatena, = asigna, ==, !=, >, <, >=, <= compara string s1 < s2 indica si alfabeticamente s1 va antes que s2. Ej: "hola"<"si"
append , +=	append characters and strings onto a string (añade una cadena al final de la cadena)
assign , =	give a string values from strings of characters and other C++ strings
begin	returns an iterator to the beginning of the string
c_str	Convierte un string en un char * (returns a standard C character array version of the string)
capacity	returns the number of elements that the string can hold (almacenar)
clear , s1=""	removes all elements from the string (elimina todos los caracteres de la cadena)
empty	true if the string has no elements (true si la cadena esta vacía, false en caso contrario)
end	returns an iterator just past the last element of a string
erase	removes elements from a string (elimina los caracteres especificados de un string)
find / rfind	(devuelve el indice de la 1ª/ultima ocurrencia de la cadena buscada)
find first/last of	(devuelve el indice de la 1ª/ultima ocurrencia de alguno de los caractes indicados)
getline	Permite leer una frase del teclado
insert (pos, s)	insert characters into a string (inserta la cadena s en la posición pos)
length , size	returns the length of the string (devuelve el nº de caracteres de la cadena)
max_size	returns the maximum number of elements that the string can hold (almacenar)
rbegin	returns a reverse iterator to the end of the string
rend	returns a reverse iterator to the beginning of the string
replace (pos, n, s):	reemplaza n caracteres a partir de la posicion pos por la cadena s
resize	change the size of the string (modifica el tamaño de la cadena, conservando lo que pueda)
substr (pos, n)	Devuelve una subcadena de n elementos a partir de la posicion pos (0 es la 1ª posición)

Ejemplos de uso de métodos:

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

int main(int argc, char *argv[]) {
    string c1="hola", c2, c3="hij";
    if (c2.empty()) cout << "c2 esta vacio\n";           //c2 esta vacio
    if (c2=="") cout << "c2 esta vacio\n";              //c2 esta vacio
    if (c2.length()==0) cout << "c2 esta vacio\n";      //c2 esta vacio
    if (c1<"si") cout << c1 << " es menor que si\n";    //hola es menor que si
    cout << "Introduce una frase: ";
    getline(cin, c1);                                   //hola tío
    c2=c1; //c2.assign(c1);
    cout << c1 << " y " << c2 << endl;                 //hola tío y hola tío
    cout << "Introduce la palabra: abcd\n";
    cin >> c1; //abcd (4)
    c2=c1; //copia c1 en c2
    cout << c1 << " y " << c2 << endl;                 //abcd y abcd
    if (c1==c2) cout << "c1 y c2 son iguales\n";        //c1 y c2 son iguales
    c1=c1+'.'+"--";                                     //abcd.-- (7)
    c1+=c2; //c1.append(c2);                             //abcd.--abcd (11)
    c1.insert(4, "efg");                                 //abcdefg.--abcd (14)
    c1.insert(7, c3);                                   //abcdefghij.--abcd (17)
    cout << c1.capacity() << endl; //17
    c1.resize(c1.size()-1); //abcdefghij.--abc (16)
    cout << c1 << c1.size() << ", " << c1.capacity() << endl; //abcdefghij.--abc16, 17
    cout << c1.substr(1,3) << ", " << c1.rfind("bc") << ", " << c1.find("bc") << endl;
                                                    //bcd,14,1
    cout << c1.find_first_of("da") << ", " << c1.find_last_of("da") << endl; //0, 13
    c1.replace(c1.find("bc"), 2, "ABCD"); cout << c1 << endl; //aABCDdefghij.--abc 18
    c1.replace(0, 10, "ab"); cout << c1 << endl;         //abij.--abc (10)
    cout << c1.size() << ", " << c1.capacity() << endl;   //10, 18
    c1=""; //c1.clear();
    if (c1.empty()) cout << "c1 esta vacio\n";          //c1 esta vacio
    c1="abc"; //abc
    cout << c1.size() << ", " << c1.length() << ", " << c1.capacity() << endl; //3,3,18
    cout << c1 << " y " << c2 << endl; //abc y abcd
    if (c1!=c2) cout << "c1 y c2 son distintos\n";      //c1 y c2 son distintos
    cout << c1 << " tiene " << c1.size() << " caracteres\n"; //abc tiene 3 caracteres
    string *cad2 = new string("Otra cadena");
    c2 += " y " + *cad2 + "\n"; //abcd y Otra cadena
    cout << c2 << endl;
    delete cad2;

    string s="abc";
    char cad[20];
    strcpy(cad, s.c_str()); //copia en cad el string y añade el '\0' final
    strcat(cad, "de");
    s=cad; //convierte el char * de C en un string
    cout << cad << ", " << s << endl;

    string::reverse_iterator e; //e es un iterador
    for(e=s.rbegin(); e!= s.rend(); e++)
        cout << *e; //muestra la cadena en orden inverso //edcba

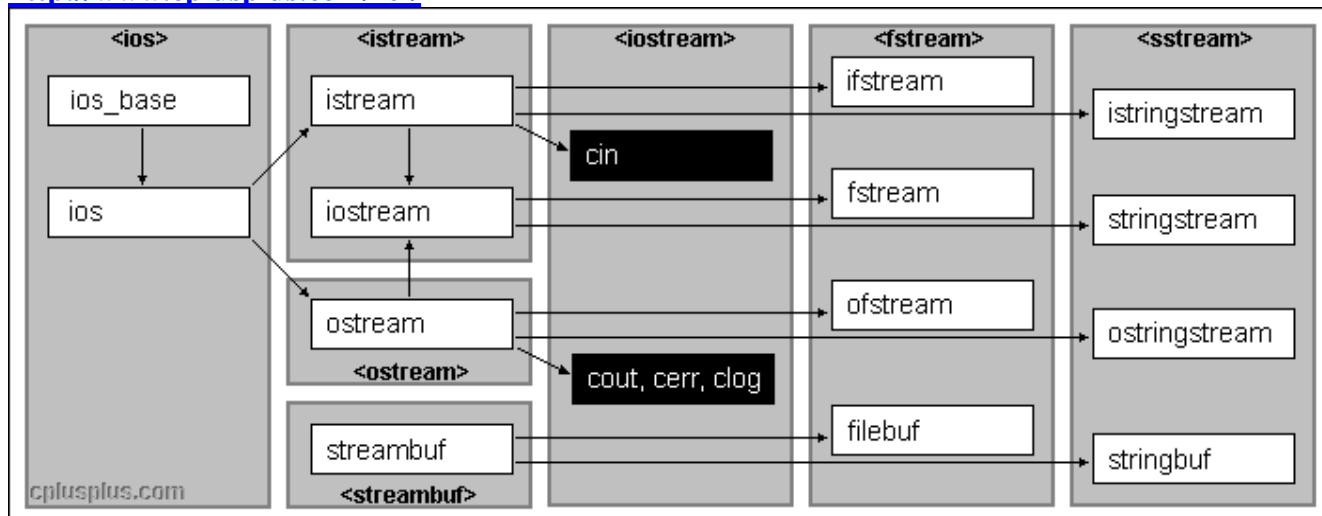
    string::iterator i; //e es un iterador
    for(i=s.begin(); i!= s.end(); i++)
        cout << *i; //muestra la cadena en orden inverso //abcde

    for(int j=s.size()-1; j>=0; j--)
        cout << s[j]; //muestra la cadena en orden inverso //edcba
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Entrada y salida con iostream (stream o flujo: fuente y/destino de datos)

La biblioteca estándar de C++ nos provee de una útil batería de clases de entrada y salida utilizando flujos. Puede consultar la jerarquía de clases en Internet:

<http://www.cplusplus.com/ref/>



En `iostream` tenemos las clases base para flujos de entrada y salida y los flujos predefinidos `cout`, `cin`, `cerr`, `clog`.

Un *stream* o *flujo* se puede definir como un dispositivo que produce o consume información. Un *flujo* está siempre ligado a un dispositivo físico. Todos los *flujos*, independientemente del dispositivo físico al que se dirijan (disco, monitor...,) se comportan de forma similar.

Un flujo es un objeto que hace de intermediario entre el programa y el origen o destino de la información. Para que un programa pueda obtener/escribir información desde/hacia un origen tiene que abrir un flujo y leer/escribir la información

Para trabajar con streams hay que importar la librería `<iostream>`, que consta de un conjunto de clases que permiten tanto la E/S por terminal (pantalla y teclado), como por ficheros y strings.

Al ejecutarse un programa en C++ se abren automáticamente los *flujos* siguientes:

cin: un flujo desde la entrada estándar (teclado) → `cin` es un objeto de la clase `istream`

cout: un flujo hacia la salida estándar (pantalla) → `cout` es un objeto de la clase `ostream`

cerr: un flujo hacia la salida de mensajes de error (pantalla) → `cerr` es un objeto de la clase `ostream`

C++ dispone de dos jerarquías de clases para las operaciones de E/S: una de bajo nivel, *streambuf*, que no se va a explicar porque sólo es utilizada por programadores expertos, y otra de alto nivel, con las clases: *istream*, *ostream* e *iostream*, que derivan de la clase *ios*. Estas clases disponen de *variables* y *métodos* para controlar los *flujos* de entrada y salida (ver la jerarquía de clases de E/S de arriba).

La clase `istream` permite leer datos de un flujo (stream) y la clase `ostream`, escribir datos en el flujo.

La clase `iostream` deriva de `istream` y `ostream` y permite leer y escribir datos en un flujo (stream)..

La clase `ifstream` deriva de `istream` y permite leer datos de un fichero.

La clase `ofstream` deriva de `ostream` y permite escribir datos en un fichero.

La clase `fstream`, que deriva de `iostream` permite escribir y leer datos de un fichero.

La clase `stringstream`, que deriva de `iostream` permite definir flujos de E/S vinculados con cadenas de caracteres

Al heredar, las clases hijas heredan todos los métodos definidos en las clases padres. La clase `ios_base` es la clase base para todas las clases que definen flujos de E/S. Todas las demás clases, al heredar de ella, pueden invocar los métodos públicos de ésta.

FUNCIONES MIEMBRO DE IOSTREAM #include <iostream>

La clase **iostream** proporciona la funcionalidad necesaria para acceder secuencial o **aleatoriamente** a un fichero abierto para leer o escribir en modo texto o **binario**. Para utilizarla: **#include <iostream>**

ostream& put(char c);	escribe el carácter c en el flujo (stream) de salida
istream& putback(char c);	devuelve al flujo el carácter c . Retorna stream de entrada
int get();	lee (extrae) el siguiente carácter del flujo de entrada y lo retorna. Retorna EOF si esta vacío o fin fichero
int peek();	idem a get() pero sin extraerlo (lo retorna pero no lo extrae)
istream& unget();	Devuelve al flujo el último carácter extraído. Retorna el stream de entrada
istream& get(char& c);	lee (extrae) el siguiente carácter del flujo de entrada y lo devuelve en el argumento c pasado por referencia. Retorna el stream de entrada
istream& get(char* s, int n, char t='\\n');	lee (extrae) los siguientes n-1 caracteres del stream de entrada y los introduce en s o hasta que encuentra el carácter de terminación t (por defecto '\\n'), o el fin de fichero. El carácter t ni se extrae ni se almacena en s , pero sí un '\\0' final en s . Retorna el stream de entrada. Falla si no se extrae ningún carácter.
istream& getline(char* s, int n, char t='\\n');	lee los siguientes n-1 caracteres del flujo de entrada o hasta que encuentra el carácter de terminación t (por defecto '\\n') o hasta el fin de fichero. Retira el carácter t del flujo de entrada, pero no lo almacena. Idem a get pero extrae t y no lo almacena
istream& ignore(int n=1, int t=eof());	extrae y descarta los siguientes n caracteres del flujo de entrada o hasta que extraiga el carácter de terminación t (por defecto EOF), o el fin de fichero
int gcount();	retorna el número de caracteres extraídos en la última extracción (no realizada con >>)
ostream& flush();	vuelca el contenido del stream (vacía el stream) al fichero vinculado al mismo
ostream& write(const char* s, int n);	escribe n bytes de la cadena s en el flujo de salida. Puede utilizarse para salida binaria.
istream& read(char* s, int n);	lee los siguientes n bytes del flujo de entrada y los deposita en la cadena s . Se usa para entrada binaria.
pos_type tellp();	devuelve la posición actual del puntero de escritura (relativo al comienzo del flujo o stream)
pos_type tellg();	devuelve la posición actual del puntero de lectura (relativo al comienzo del flujo o stream)
ostream& seekp(pos_type pos);	posiciona el puntero de escritura a la posición absoluta pos del fichero
ostream& seekp(desp, pos);	desplaza la posición actual de escritura desp bytes respecto a la posición pos que puede ser: ios::beg (principio del flujo) ios::cur (posición actual del puntero de escritura) ios::end (final del flujo)
istream& seekg(pos_type pos);	idem, pero para el puntero de lectura
istream& seekg(desp, pos);	idem, pero para el puntero de lectura

La mayor parte de las funciones anteriores devuelven una referencia al flujo de entrada o de salida. Esta referencia puede utilizarse para detectar errores o la condición de fin de fichero

FUNCIONES MIEMBRO DE FSTREAM #include <fstream>

La clase **fstream** es una clase derivada de **iostream** (hereda todos sus métodos) que permite poder leer desde o escribir en ficheros (lectura/escritura de datos en unidades de almacenamiento permanente como los disquetes, discos duros, etc.). Para utilizarla: **#include <fstream>**

fstream();	constructor por defecto de la clase. Construye un flujo sin abrir ningún fichero. El fichero puede ser abierto más tarde con la función <i>open()</i> .
fstream(const char* f, int modo=ios::in ios::out);	constructor que crea un objeto fstream y abre un fichero f con el modo de apertura modo (por defecto ios::in ios::out es decir lectura y escritura).
~fstream();	Destructor que vuelca el buffer del fichero y cierra el fichero, si no está cerrado ya.
void open(const char* f, int modo=ios::in ios::out);	abre el fichero f con el modo de apertura modo (por defecto ios::in ios::out). Sus argumentos son los mismos que los de <i>fstream()</i> .
bool isopen();	retorna true si el fichero esta abierto
void close();	cierra el fichero asociado si no está cerrado ya.

Los modos de apertura de un fichero son los siguientes:

ios::in	Abre en modo lectura (permite solo operaciones de lectura)
ios::out	Abre en modo escritura (permite solo operaciones de escritura)
ios::ate	Va al final del stream en la apertura (colocarse al final del fichero)
ios::app	(append). Añade datos al final del fichero
ios::trunc	(truncate). Borra el contenido previo del fichero al abrir si ya existe. Es el valor por defecto si solo se indica out
ios::binary	Abre en modo binario. Los caracteres \r y \n no son convertidos. Se ha de activar cuando se trabaje con ficheros de datos binarios.

Los modos en los que se puede abrir un fichero son los siguientes:

Modo texto (por omisión)		Modo binario
ios::in	abierto para lectura	ios::in ios::binary
ios::out	abierto para escritura Si el fichero existe se destruye su contenido	ios::out ios::binary
ios::out ios::trunc	abierto para escritura Si el fichero existe se destruye su contenido	ios::out ios::trunc ios::binary
ios::out ios::app	Añadir al final No se destruye su contenido	ios::out ios::app ios::binary
ios::in ios::out	Lectura/escritura	ios::in ios::out ios::binary
ios::out ios::trunc ios::in	Lectura/escritura Si el fichero existe se destruye su contenido	ios::out ios::trunc ios::in ios::binary

Antes de **abrir un fichero** hay que crear un **flujo**, es decir un objeto de las clases **ifstream**, **ofstream** o **fstream** e indicar el modo de apertura (lectura, escritura, lectura y escritura, ...).

Por ejemplo para abrir un fichero para lectura de datos creando un **fstream fichero**:

```
fstream fichero;                                ifstream fichero;
fichero.open("datos.dat", ios::in);             fichero.open("datos.dat");
```

y para escritura en ese mismo fichero:

```
fstream fichero;                                ofstream fichero;
fichero.open("datos.dat", ios::out);             fichero.open("datos.dat");
```

ifstream, **ofstream** y **fstream** tienen constructores que permiten abrir ficheros de forma automática

```
ifstream fichero("datos.dat");
```

donde se sobreentiende que el fichero se abre para **lectura** por haber utilizado **ifstream**. Si se hubiese utilizado **ofstream** el fichero se hubiera abierto para escritura.

Un fichero se puede abrir en modo texto o modo binario

Cadenas (modo texto)

```
int main() {
    char cadena[128];
    ofstream fs("nombre.txt"); //abre nombre.txt para escritura
    fs << "Hola, mundo" << endl;
    fs.close(); //cierra el fichero
    ifstream fe("nombre.txt"); //abre nombre.txt para lectura
    fe.getline(cadena, 128);
    cout << cadena << endl;
}
```

Binarios (ejemplo de copia de fichero en modo binario)

```
int main() {
    char Desde[] = "excepcion.cpp"; // Este fichero
    char Hacia[] = "excepcion.cpy";
    char buffer[1024];
    int leido;
    ifstream fe(Desde, ios::binary);           //fstream fe(Desde, ios::in | ios::binary);
    ofstream fs(Hacia, ios::binary);           //fstream fs(Hacia, ios::out | ios::binary);

    do {
        fe.read(buffer, 1024);
        leido = fe.gcount();                    //gcount devuelve el numero de caracteres (bytes) leidos
        fs.write(buffer, leido);
    } while(leido);
    fe.close();
    fs.close();
    cout << " fin de copia";

    cin.get();
    return 0;
}
```

Funciones clave

open, close, read, write, get, getline, fail, good, exceptions, eof,

ERRORES DE E/S. ESTADOS DE UN FLUJO

Al leer y escribir flujos (ficheros, consola, etc.) se pueden producir errores (no encontrar el fichero o no poderlo abrir, o al menos situaciones de excepción, tales como el haber llegado al fin del fichero).

Cada flujo de E/S tiene asociado un estado que la clase *ios* define en una variable enum **io_state** con los siguientes valores: goodbit, eofbit, badbit y failbit. Cada flujo de E/Sa mantiene información sobre los errores que se hayan podido producir. Esta información se puede chequear con las siguientes funciones:

<i>int good ();</i>	devuelve un valor distinto de cero (true) si no ha habido errores (si todos los bits de error están en off, es decir a 0). Indica que la ultima operación ha tenido éxito
<i>int eof();</i>	devuelve un valor distinto de cero si se ha llegado al fin del fichero.
<i>int bad();</i>	devuelve un valor distinto de cero si ha habido un error de E/S serio. No se puede continuar en esas condiciones.
<i>int fail();</i>	devuelve un valor distinto de cero si ha habido cualquier error de E/S distinto de EOF. Si una llamada a <i>bad()</i> devuelve 0 (no error de ese tipo), el error puede no ser grave y la lectura puede proseguir después de llamar a la función <i>clear()</i> .
<i>int clear();</i>	se borran los bits de error que pueda haber activados.

Además, tanto los operadores sobrecargados (<< y >>) como las funciones miembro de E/S devuelven referencias al flujo correspondiente y esta referencia puede chequearse con un ***if*** o en la condición de un ***while*** para saber si se ha producido un error o una condición de fin de fichero. Por ejemplo, las siguientes construcciones pueden utilizarse en C++:

```
while (cin.get(ch)) {           while (cin << ch) {
    s[i++] = ch;                  s[i++] = ch;
}
```

ya que si el valor de retorno de (***cin.get(ch)***) o de (***cin<<ch***) no es nulo, es que no se ha producido error ni se ha llegado al fin de fichero.

Si lo que se quiere comprobar es si se ha llegado al final del fichero se puede comprobar la condición,

```
if (cin.get(ch).eof()) {
    // se ha llegado al final del fichero
}
```

y si lo que se desea es hacer algo si no se ha tenido ningún error, se puede utilizar el ***operador negación (!)*** que devuelve un resultado distinto de cero (true) si ***failbit*** o ***badbit*** están activados.

```
if (!cin.get(ch)) {
    // hay un error de tipo failbit o badbit.
}
```

El operador negación se puede utilizar también en la forma siguiente, para saber si un fichero se ha podido abrir correctamente:

```
fstream filein("datos.d", ios::in);
if (!filein) {
    cerr << "no se ha podido abrir el fichero." << endl;
    exit(1);
}
```

Public member functions: (www.cplusplus.com)

stringstream members:

(constructor)	Construct an object and optionally initialize string content.
rdbuf	Get the stringbuf object associated with the stream.
str	Get/set string value.

fstream members:

(constructor)	Construct an object and optionally open a file.
rdbuf	Get the filebuf object associated with the stream.
is_open	Check if a file has been opened.
open	Open a file.
close	Close an open file.

members inherited from istream:

operator>>	Performs a formatted input operation (extraction)
gcount	Get number of characters extracted by last unformatted input operation
get	Extract unformatted data from stream
getline	Get a line from stream
ignore	Extract and discard characters
peek	Peek next character
read	Read a block of data
readsome	Read a block of data
putback	Put the last character back to stream
unget	Make last character got from stream available again
tellg	Get position of the get pointer
seekg	Set position of the get pointer
sync	Synchronize stream's buffer with source of characters

members inherited from ostream:

operator<<	Perform a formatted output operation (insertion).
flush	Flush buffer.
put (char)	Inserta un caracter en el output stream (stream de salida).
seekp	Set position of put pointer.
tellp	Get position of put pointer.
write	Write a sequence of characters.

members inherited from ios:

operator void *	Convert stream to pointer.
operator !	evaluate stream object.
bad	Check if an unrecoverable error has occurred.
clear	Set control states.
copyfmt	Copy formatting information.
eof	Check if End-Of-File has been reached.
exceptions	Get/set the exception mask.
fail	Check if failure has occurred.
fill	Get/set the fill character.
good	Check if stream is good for i/o operations.
rdbuf	Get/set the associated streambuf object.
rdstate	Get control state.
setstate	Set control state.
tie	Get/set the tied stream.
widen	Widen character.

members inherited from ios_base:

flags	Get/set format flags.
getloc	Get current locale.
imbue	Imbue locale.
setf	Set some format flags.

Entrada y salida con sstream

sstream permite utilizar **cadenas** como flujos.

Pantalla

```
flujoCadena:
+ABCDa 1
    linea 2
5
FIN

strC1:
    Linea 1
strC2:
    linea
x:
    2
strCPP:
    5
cadena:
    ABCDa

cad:
+ABCDa 1
    linea 2
5
FIN

Presione una tecla para
continuar . . .
```

Utilidad:

Si queremos devolver en un `char *` o en un `string` una cadena determinada, podemos utilizar `stringstream` para formar la cadena como si estuviéramos escribiéndola en pantalla con `cout` y luego con el método `str()` pasarla a un `string` y de éste a un `char *` con el método `c_str()`

```
#include <cstdlib>
#include <iostream> //clase que deriva de istream y ostream
#include <sstream> //para usar stringstream
#include <string> //para usar string

using namespace std;

int main(int argc, char *argv[])
{
    stringstream flujoCadena; //uso flujoCadena como si fuera cin o cout
    char strC1[200], strC2[200];
    int x, dato=5;
    string strCPP, cadena;
    /* Salida a la cadena */
    flujoCadena << " Linea 1\n linea 2\n" << dato << endl << "FIN\n";
    /* Leer una línea (no ignora los blancos iniciales y no añade \n) */
    flujoCadena.getline(strC1,200); //lee un máximo de 200 char
    // Con el operador >> se puede leer una palabra (descarta los blancos)
    flujoCadena >> strC2; //como cin hacia un char *
    //Leer la siguiente entrada. Descarta los blancos iniciales por defecto... */
    flujoCadena >> x;
    flujoCadena >> strCPP; //Leer la siguiente entrada

    //podemos usar metodos de la clase istream ya que deriva de ella
    flujoCadena.seekg(1); //posiciona el puntero de lectura en la pos 1
    flujoCadena.seekp(0); //posiciona el puntero de escritura al principio
    flujoCadena.put('+'); //añade el caracter + en la posicion 0
    flujoCadena << "ABCD"; //añade ABCD a continuacion
    flujoCadena >> cadena;

    /* El método .str() nos da un string el contenido del flujo */
    cout << "flujoCadena:\n" << flujoCadena.str() << "\n";
    cout << "strC1:\n" << strC1 << "\n";
    cout << "strC2:\n" << strC2 << "\n";
    cout << "x:\n" << x << "\n";
    cout << "strCPP:\n" << strCPP << "\n";
    cout << "cadena:\n" << cadena << "\n";

    string s=flujoCadena.str();
    char cad[200];
    strcpy(cad,flujoCadena.str().c_str());
    cout << "\ncad:\n";
    cout << cad << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Entrada y salida con iostream

fstream nos permite utilizar **archivos** como flujos.

Los **modos de apertura** son constantes de **máscara de bit**, de modo que se puede hacer un *or* lógico de ellos para conseguir un modo de apertura combinado.

Ejemplo en modo texto:

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

int main(int argc, char *argv[]) {
    // fstream archivo( "Prueba.txt", fstream::out | fstream::trunc );
    // fstream archivo;
    /*Abrimos archivo en modo salida y borramos su contenido*/
    archivo.open( "Prueba.txt", ios::out | fstream::trunc );
    if ( archivo.is_open() ) { // Comprobamos que se abrió bien
        archivo << "Juan Jose, Luis, " << 5 << endl; //escribimos
        archivo.close(); //cerramos
    }

    /* Abrir el archivo para escritura y añadir al final */
    archivo.open( "Prueba.txt", ios::out | ios::app );
    if (archivo.is_open()) {
        archivo << "Marco Antonio, Pedro " << 4 << endl;
        archivo.close();
    }

    /* Abrir el archivo en modo entrada y leer con operator >> */
    archivo.open( "Prueba.txt", ios::in );
    if ( archivo.is_open() ) { //true si el fichero esta abierto
        string lectura;
        cout << "tellg: " << archivo.tellg() << endl;
        /* Para controlar fin de archivo correctamente, hay que hacer
        una lectura antes de comprobar si se ha llegado al EOF */
        archivo >> lectura;
        cout << "--good: " << archivo.good() << endl;
        while ( !archivo.eof() ) {
            cout << lectura;
            archivo >> lectura;
            cout << "--good: " << archivo.good() << endl;
        }
        //al salirse del while el bit eofbit estara a 1-->
        //good() estara a false porque la ultima operacion no
        //ha tenido exito -> invocar clear() para borrar bits de error
        cout << "FIN --good: " << archivo.good();
        cout << " tellg: " << archivo.tellg() << endl;
        if (!archivo.eof() && archivo.fail())
            cerr << "Error al leer del fichero\n";
        archivo.clear(); //MUY IMPORTANTE
        archivo.close(); //cierra el fichero
    }
}
```

El constructor de **fstream** permite indicar el fichero a abrir y el modo de apertura

Si no se limpia los bits de error el programa no funcionaria bien. Quítalos y veo lo que ocurre

```
/* Abrimos el archivo en modo entrada y leer con getline */
/* mostrar las lineas en pantalla numeradas y los bytes leidos */
archivo.open( "Prueba.txt", ios::in );
int i=1;
if ( archivo.is_open() ) { //true si el fichero esta abierto
    char texto[100];
    archivo.getline(texto,100); //lee una linea o 99 char max
    cout << i++ << ": " << texto << " (" << archivo.gcount()
    << ") \n";
    while ( !archivo.eof() ) {
        archivo.getline(texto,100);
        cout << i++ << ": " << texto << " (" <<
        archivo.gcount() << ") \n";
    }
    archivo.clear(); //MUY IMPORTANTE
    archivo.close(); //cierra el fichero
}

/* Abrir el archivo y mostrar cada nombre en una linea */
archivo.open( "Prueba.txt", ios::in );
if ( archivo.is_open() ) {
    char texto[100];
    //lee hasta el char ',' o '\n'
    while (archivo.getline(texto,100, ',') != NULL)
        cout << texto << endl;
    archivo.clear(); //MUY IMPORTANTE
    archivo.close(); //cierra el fichero
}

system("PAUSE");
return EXIT_SUCCESS;
}
```

Pantalla:

```
tellg: 0 --good: 1
Juan--good: 1
Jose,--good: 1
Luis,--good: 1
5--good: 1
Marco--good: 1
Antonio,--good: 1
Pedro--good: 1
4--good: 0
FIN --good: 0 tellg: -1
```

```
1:Juan Jose, Luis, 5 (19)
2:Marco Antonio, Pedro 4 (23)
3: (0)
```

```
Juan Jose
Luis
5
Marco Antonio
Pedro 4
```

Presione una tecla para continuar . . .

Modos de apertura

in	Abrir para lectura
out	Abrir para escritura
ate	Colocarse al final del fichero
app	Añadir al final del fichero
trunc	Borra el contenido, si existe
binary	Modo binario

Entrada y salida con iostream (fstream, ejemplo en modo binario con estructuras)

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

struct registro{
    char nombre[21];
    char nif[10];
    int edad;
};

int main(int argc, char *argv[]) {
    registro r1 = { "Jose Jose","75555587F", 20 };
    /* Abrimos el archivo en modo salida y descartamos contenido actual */
    fstream archivo( "Prueba.bin", ios::out | ios::trunc | ios::binary );
    archivo.write( (char*)&r1, sizeof( registro ) );
    archivo.close();

    strcpy( r1.nombre, "Ana Marta" ); /* Otro registro */
    strcpy( r1.nif, "75555586X" );
    r1.edad = 19;

    /* Abrimos el archivo en modo salida y añadir al final */
    archivo.open( "Prueba.bin", fstream::out | ios::app | ios::binary );
    archivo.write( (char*)&r1, sizeof( registro ) );
    cout << "Desea introducir datos (s/n)? " << endl;
    char respuesta;
    cin >> respuesta; //si tecleo s o ss o sloquesea no hay problema porque
                      //la linea siguiente extrae todo lo que quede en buffer teclado
    cin.ignore(numeric_limits<int>::max(), '\n'); //entrae todo lo que haya hasta
                      //encontrar un \n (el \n lo extrae)

    while (respuesta=='s') {
        cout << "Nombre: "; cin.getline(r1.nombre, 21); //puede tener blancos
        cout << "NIF: "; cin >> r1.nif; //no tiene blancos
        cout << "edad: "; cin >> r1.edad;
        archivo.write( (char*)&r1, sizeof( registro ) );
        cout << "Desea introducir datos (s/n)? " << endl;
        cin >> respuesta;
        cin.ignore(); //extrae un caracter (el \n que queda en el buffer del teclado)
                      //si en vez de teclear s tecleo ssss --> problema!!!
    }
    archivo.close();

    /* Abrimos el archivo en modo entrada */
    archivo.open( "Prueba.bin", fstream::in | fstream::binary );
    registro r;
    /* Para controlar el fin de archivo correctamente, es necesario
       hacer una lectura antes de comprobar si se ha llegado a EOF */
    archivo.read( (char*)&r, sizeof( registro ) );
    while (!archivo.eof()){
        cout << r.nombre << " " << r.nif << " (" << r.edad << ")\n";
        archivo.read( (char*)&r, sizeof( registro ) );
    }
    archivo.clear(); //para que limpie el bit de error eofbit
    archivo.close();

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Se ha omitido la comprobación de apertura correcta (is_open()) por razones de espacio.

```
Desea introducir datos (s/n)?
ssssssss
Nombre: eva maria
NIF: 123456789K
edad: 26
Desea introducir datos (s/n)?
s
Nombre: juan
NIF: 987654321A
edad: 45
Desea introducir datos (s/n)?
n
Jose Jose 75555587F (20)
Ana Marta 75555586X (19)
eva maria 123456789K (26)
juan 987654321A (45)
Presione una tecla para
continuar . . .
```

Entrada y salida con iostream (fstream, ejemplo en modo binario con clases)

Cuando la clase no tiene memoria dinámica todos los objetos de la clase tienen el mismo tamaño que es constante (escribimos y leemos copia binaria de los datos):

Guardar la clase en fichero: `archivo.write((char*)this, sizeof(NombreClase));`

Recuperar la clase de fichero: `archivo.read((char*)this, sizeof(NombreClase));`

PODEMOS TRABAJAR COMO SI FUERA CON ESTRUCTURAS EN VEZ DE CON CLASES (no importa que tenga parte privada, ya que los métodos write y read trabajan a nivel de bytes y se ‘saltan’ dichas restricciones de acceso)

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

class Provincias {
private:
    char nombre[21];
    char capital[21];
    int censo;
public:
    void pinta() { cout << nombre << " " << capital << " " << censo << endl; }
    Provincias( char *nom, char *cap, int cen ) { //constructor
        strcpy( nombre, nom );
        strcpy( capital, cap );
        censo = cen;
    }
};

int main(int argc, char *argv[])
{
    fstream archivo;
    Provincias r1("Huelva","Huelva",150000),r2("Mallorca","Palma de Mallorca",250000);
    archivo.open( "Prueba.bin", ios::out | ios::trunc | ios::binary );
    archivo.write( (char*)&r1, sizeof( Provincias ) );
    archivo.close();
    //Añadir
    archivo.open( "Prueba.bin", fstream::out | fstream::app | fstream::binary );
    archivo.write( (char*)&r2, sizeof( Provincias ) );
    archivo.close();

    archivo.open( "Prueba.bin", fstream::in | fstream::binary );
    archivo.read( (char*)&r1, sizeof( Provincias ) );
    while (!archivo.eof()){
        r1.pinta();
        archivo.read( (char*)&r1, sizeof( Provincias ) );
    }
    archivo.close();

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Se ha omitido la comprobación de apertura correcta (`is_open()`) por razones de espacio.

Cuando la clase no tiene memoria dinámica podemos utilizar directamente en el main write y read haciendo una copia binaria de los datos (sobreescribiendo incluso la parte privada):

`archivo.write((char*)&objeto, sizeof(Clase));`
`archivo.read((char*)&objeto, sizeof(Clase));`

escritura binaria y lectura binaria de los datos directamente en el main

Entrada y salida con iostream (fstream, ejemplo en modo binario con clases)

Cuando la clase no tiene memoria dinámica todos los objetos de la clase tienen el mismo tamaño que es constante (escribimos y leemos copia binaria de los datos):

Guardar la clase en fichero: `archivo.write((char*)this, sizeof(NombreClase));`

Recuperar la clase de fichero: `archivo.read((char*)this, sizeof(NombreClase));`

LO HABITUAL ES QUE DEFINAMOS EN LA CLASE UN METODO PARA GUARDAR EL OBJETO EN FICHERO Y OTRO PARA RECUPERAR LA INFORMACION DEL OBJETO DE FICHERO

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

```
class Registro {
    char nombre[21];
    int edad;
public:
    void pinta(){ cout << nombre << " " << edad << endl; }
    void cambia( char *nom, int e ){ strcpy( nombre, nom ); edad = e; }
    void almacenar( fstream &archivo ){ archivo.write( (char*)this, sizeof( Registro ) ); }
    void recuperar( fstream &archivo ){ archivo.read( (char*)this, sizeof( Registro ) ); }
};
```

```
int main(int argc, char *argv[]) {
    fstream archivo;
    Registro r1,r2;
    r1.cambia("Jose Perez",20); r2.cambia("Pepe",27);
    archivo.open( "Prueba.bin", fstream::out | fstream::trunc | fstream::binary );
    r1.almacenar( archivo ); //almaceno el objeto r1 en el archivo
    r2.almacenar( archivo ); //almaceno el objeto r2 en el archivo
    archivo.close();
```

```
    r1.cambia("Ana",19);
    archivo.open( "Prueba.bin", fstream::out | fstream::app | fstream::binary );
    r1.almacenar( archivo );
    archivo.close();
```

```
    archivo.open( "Prueba.bin", fstream::in | fstream::binary );
    r1.recuperar( archivo );
    while ( !archivo.eof() ) {
        r1.pinta();
        r1.recuperar( archivo );
    }
```

```
    archivo.clear(); //para que limpie el bit de error eofbit //quita esta línea y vea lo que ocurre...
    archivo.close();
```

//ejemplo de acceso aleatorio

```
    archivo.open( "Prueba.bin", fstream::in | fstream::binary );
    archivo.seekg(0, ios::end); //me pongo al final del fichero
    cout << "En el fichero hay " << (float) archivo.tellg() / (int)sizeof( Registro ) << " registros\n";
    archivo.seekg(0, ios::beg); //me pongo en el primer registro
    cout << "primero: "; r1.recuperar( archivo ); r1.pinta(); //lo recupero y lo muestro en pantalla
```

```
    archivo.seekg(-(int)sizeof( Registro ), ios::end); //me pongo al inicio del ultimo registro (int) VIP
    cout << "ultimo: "; r1.recuperar( archivo ); //tras leer el puntero lectura avanza sizeof( Registro ) bytes
    r1.pinta();
    archivo.clear(); //para que limpie el bit de error eofbit
    archivo.close();
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Se ha omitido la comprobación de apertura correcta (`is_open()`) por razones de espacio.

Hay que pasar el archivo por referencia.

Cuando la clase no tiene memoria dinámica:

```
archivo.write( (char*)this, sizeof( Clase ) );
archivo.read( (char*)this, sizeof( Clase ) );
```

escritura binaria y lectura binaria de los datos

```
Jose Perez 20
Pepe 27
Ana 19
```

En el fichero hay 3 registros
primero: Jose Perez 20
ultimo: Ana 19

Quitando la línea amarilla...

```
Jose Perez 20
Pepe 27
Ana 19
En el fichero hay -0.0357143
registros
primero: Ana 19
ultimo: Ana 19
```

Porque al no limpiar los bits de error lo gris no se ejecutan porque el puntero de lectura esta en eof (tellg() devuelve -1) → r1 no ha cambiado

Entrada y salida con iostream (fstream, ejemplo en modo binario con clases)

Cuando la clase tiene memoria dinámica es mucho mas complejo (no podemos hacer copia binaria de los datos) ya que **todos los objetos de la clase NO tienen el mismo tamaño y el espacio de memoria dinámica reservado por dicha clase no es continuo**, pudiendo haber multiples referencias a otros objetos. Es necesario guardar la información de manera que sea posible recuperar y montar la estructura partiendo sólo de la información disponible en el fichero.

LO HABITUAL ES QUE DEFINAMOS EN LA CLASE UN METODO PARA GUARDAR EL OBJETO EN FICHERO Y OTRO PARA RECUPERAR LA INFORMACION DEL OBJETO DE FICHERO

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

**En Dev-C++ hay
que ejecutar el .exe
para poder
visualizarlo...!!!**

```
class Registro{
    char *nombre;
    int edad;
public:
    Registro() { nombre=NULL; edad=0; }
    ~Registro() { if (nombre!=NULL)
        delete [] nombre; }

    void pinta(){
        if (nombre!=NULL)
            cout << nombre << " " << edad << endl;
    }

    void cambia( char *nom, int e ) {
        if (nombre!=NULL) delete [] nombre;
        nombre=new char[strlen(nom)+1];
        strcpy( nombre, nom );
        edad = e;
    }

    bool almacenar( fstream &f ){
        //archivo.write( (char*)this, sizeof( Registro ) );
        int n=strlen(nombre)+1;
        f.write((char *)(&n),sizeof(int));
        f.write((char *)(&nombre),n);
        f.write((char *)(&edad),sizeof(int));
        return(f.good());
    }

    bool recuperar(fstream &f){
        //archivo.read( (char*)this, sizeof( Registro ) );
        int n;
        f.read((char *)(&n),sizeof(int));
        if (nombre!=NULL) delete [] nombre;
        nombre=new char [n];
        f.read((char *)(&nombre),n);
        f.read((char *)(&edad),sizeof(int));
        return(f.good());
    }
};
```

```
Jose Perez 20
Pepe 27
Ana 19
posbak: 32 pos: 44
En el fichero hay 3 registros
primero: Jose Perez 20      32
ultimo: Ana 19
Quitando la linea amarilla...
Jose Perez 20
Pepe 27
Ana 19
posbak: 0 pos: 0
En el fichero hay 0 registros
primero: 19                -1
ultimo: 19
```

```
int main(int argc, char *argv[]) {
    fstream archivo;
    Registro r1,r2;
    r1.cambia("Jose Perez",20); r2.cambia("Pepe",27);
    archivo.open( "Prueba.bin", ios::out | ios::trunc | ios::binary );
    r1.almacenar( archivo ); r2.almacenar( archivo );
    archivo.close();

    r1.cambia("Ana",19);
    archivo.open( "Prueba.bin", ios::out | ios::app | ios::binary );
    r1.almacenar( archivo );
    archivo.close();

    archivo.open( "Prueba.bin", fstream::in | fstream::binary );
    r1.recuperar( archivo );
    while (!archivo.eof()) {
        r1.pinta();
        r1.recuperar( archivo );
    }
    archivo.clear(); //para que limpie el bit de error eofbit //quita
    esta linea y vea lo que ocurre...
    archivo.close();

    //ejemplo de acceso aleatorio
    archivo.open( "Prueba.bin", fstream::in | fstream::binary );
    //archivo.seekg(0, ios::end); //me pongo al final del fichero
    //cout << "En el fichero hay " << (float) archivo.tellg() / (int) sizeof(
    Registro ) << " registros\n";

    int n=0, pos=0, posbak=0;
    r1.recuperar( archivo );
    while (!archivo.eof()) {
        n++; //voy contando los registros que hay
        posbak=pos;
        pos=archivo.tellg(); //guardo la posicion puntero lectura
        r1.recuperar( archivo );
    }
    cout << "posbak: " << posbak << " pos: " << pos << endl;
    archivo.clear(); //VIP al salir del while hay un error en el bit eof
    cout << "En el fichero hay " << n << " registros\n";
    archivo.seekg(0, ios::beg); //me pongo en el primer registro
    cout << "primero: "; r1.recuperar( archivo ); r1.pinta();
    //archivo.seekg( (int) sizeof( Registro ), ios::end); //ultimo registro
    archivo.seekg(posbak, ios::beg); //me pongo en el ultimo registro
    cout << archivo.tellg() << endl;

    cout << "ultimo: "; r1.recuperar( archivo );
    //tras leer el puntero lectura avanza sizeof( Registro ) bytes
    r1.pinta();
    archivo.clear(); //para que limpie el bit de error eofbit
    archivo.close();
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Entrada y salida con iostream (fstream, ejemplo en modo binario con clases)

Cuando la clase tiene memoria dinámica es mucho mas complejo (no podemos hacer copia binaria de los datos) ya que **todos los objetos de la clase NO tienen el mismo tamaño y el espacio de memoria dinámica reservado por dicha clase no es continuo**, pudiendo haber multiples referencias a otros objetos. Es necesario guardar la información de manera que sea posible recuperar y montar la estructura partiendo sólo de la información disponible en el fichero.

LO HABITUAL ES QUE DEFINAMOS EN LA CLASE UN METODO PARA GUARDAR EL OBJETO EN FICHERO Y OTRO PARA RECUPERAR LA INFORMACION DEL OBJETO DE FICHERO

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

class Provincias {
    char *nombre;
    char *capital;
    int censo;
public:
    void pinta() { cout << nombre << " " << capital << " " << censo << endl; }
    Provincias( char *nom, char *cap, int cen ) { //constructor
        nombre=new char[strlen(nom)+1]; strcpy( nombre, nom );
        capital=new char[strlen(cap)+1]; strcpy( capital, cap );
        censo = cen;
    }
    ~Provincias () { delete [] nombre; delete [] capital; }

    bool almacenar( fstream &f ){
        int n1=strlen(nombre)+1, n2=strlen(capital)+1;
        f.write((char *)&n1,sizeof(int)); f.write((char *)nombre,n1); //guardo el tamaño y nombre
        f.write((char *)&n2,sizeof(int)); f.write((char *)capital,n2); //guardo el tamaño y capital
        f.write((char *)&censo,sizeof(int)); //guardo censo
        return(f.good()); //true si se ha escrito bien
    }

    bool recuperar(fstream &f){
        int n1, n2;
        f.read((char *)&n1,sizeof(int)); //recupero tamaño
        delete [] nombre; delete [] capital; //libero memoria
        nombre=new char [n1]; f.read((char *)nombre,n1);
        f.read((char *)&n2,sizeof(int)); //recupero tamaño
        capital=new char [n2]; f.read((char *)capital,n2);
        f.read((char *)&censo,sizeof(int));
        return(f.good());
    }
};

int main(int argc, char *argv[]) {
    fstream archivo;
    Provincias r1("Huelva","Huelva",150000),r2("Mallorca","Palma de Mallorca",250000);
    archivo.open( "Prueba.bin", ios::out | ios::trunc | ios::binary );
    r1.almacenar( archivo ); //archivo.write((char*)&r1, sizeof( Provincias ));
    archivo.close();
    archivo.open( "Prueba.bin", fstream::out | fstream::app | fstream::binary );
    r2.almacenar( archivo ); //archivo.write((char*)&r2, sizeof( Provincias ));
    archivo.close();
    archivo.open( "Prueba.bin", fstream::in | fstream::binary );
    r1.recuperar( archivo ); //archivo.read((char*)&r1, sizeof( Provincias ));
    while (!archivo.eof()){
        r1.pinta();
        r1.recuperar( archivo ); //archivo.read((char*)&r1, sizeof( Provincias ));
    }
    archivo.close(); system("PAUSE"); return EXIT_SUCCESS;
}
```

Se ha omitido la comprobación de apertura correcta (is_open()) por razones de espacio.

Cuando la clase tiene memoria dinámica no podemos utilizar directamente en el main **write** y **read** haciendo una copia binaria de los datos (sobreescribiendo incluso la parte privada):

```
archivo.write((char*)&objeto, sizeof( Clase ));
archivo.read((char*)&objeto, sizeof( Clase ));
```

ya que el tamaño de cada objeto es diferente

Si se quiere guardar una estructura aún más compleja se hace necesario establecer un orden.

```
class ComunidadAutonoma {
    Provincias **lista;
    int nmax,cont;
public:
    ComunidadAutonoma(int _nmax=1) { //tamaño por defecto 1
        nmax = _nmax; //numero maximo de provincias
        cont =0; //numero de provincias actuales
        lista= new Provincias*[nmax];
    }
    void add(Provincias &c) { //& indica paso por referencia
        if (cont==nmax) {
            Provincias *l=new Provincias*[nmax+3]; //crece de 3 en 3
            for(int i=0; i<nmax; i++) l[i]=lista[i];
            delete [] lista;
            lista=l;
            nmax+=3;
        }
        lista[cont++]=&c; //& indica direccion de memoria objeto c
    }
    bool almacenar(fstream &f) {
        f.write((char *)&nmax,sizeof(int)); //guardo nmax
        f.write((char *)&cont,sizeof(int)); //guardo cont
        for (int i=0;i<cont;i++){
            lista[i]->almacenar(f); //llamo a almacenar de Provincias
            return(f.good());
        }
    }
    bool recuperar(fstream &f){
        delete [] lista; //borra la lista actual
        f.read((char *)&nmax,sizeof(int));
        f.read((char *)&cont,sizeof(int));
        lista = new Provincias*[nmax]; //reseva memoria para lista
        for (int i=0;i<cont;i++){
            Provincias *tmp= new Provincias(" ", " ", 0); //crea un objeto
            tmp->recuperar(f);
            lista[i]=tmp;
        }
        return(f.good());
    }
    void pinta() {
        for (int i=0;i<cont;i++)
            lista[i]->pinta();
    }
};
```

```
c1:
Alava Victoria 300
Guipuzcoa San Sebastian 250
Vizcaya Bilbao 800
```

```
c2:
Vizcaya Bilbao 800
Guipuzcoa San Sebastian 250
Alava Victoria 300
```

```
c2:
Vizcaya Bilbao 800
8%R 250
Ç$R P%R 300
```

```
Comunidad 1:
Alava Victoria 300
Guipuzcoa San Sebastian 250
Vizcaya Bilbao 800
```

```
Comunidad 2:
Vizcaya Bilbao 800
Guipuzcoa San Sebastian 250
Alava Victoria 300
```

```
Hay 2 Comunidades
```

```
int main(int argc, char *argv[]) {
    fstream archivo;
    ComunidadAutonoma c1(3), c2(2), c; //c(1)
    Provincias *r3; //puntero a Provincias

    {
        Provincias r1("Alava","Victoria",300);
        Provincias r2("Guipuzcoa","San Sebastian",250);
        r3=new Provincias("Vizcaya","Bilbao",800);
        c1.add(r1); c1.add(r2); c1.add(*r3);
        c2.add(*r3); c2.add(r2); c2.add(r1);
        cout << "c1:\n"; c1.pinta();
        cout << "\nc2:\n"; c2.pinta();
        archivo.open( "Prueba.bin",
                    ios::out | ios::trunc | ios::binary );
        c1.almacenar( archivo ); //c1 salvado en fichero
        archivo.close();
        archivo.open( "Prueba.bin",
                    ios::out | ios::app | ios::binary );
        c2.almacenar( archivo ); //c2 salvado en fichero
        archivo.close();
    }
    //las variables locales al bloque se destruyen

    cout << "\nc2:\n"; c2.pinta(); //r1 y r2 ya no existen
    archivo.open( "Prueba.bin", ios::in | ios::binary );
    c.recuperar( archivo );
    int i=0;
    while (!archivo.eof()) {
        cout << "\nComunidad " << ++i << ":\n";
        c.pinta();
        c.recuperar( archivo );
    }
    cout << "\nHay " << i << " Comunidades\n";
    archivo.clear();
    archivo.close();
    delete r3;
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Lo gris en c2 (y en c1 ocurriría lo mismo) se debe a que guardamos la dirección de memoria de los objetos r1, r2 y *r3. Las objetos r1 y r2 son locales al bloque, con lo que al salirse del bloque dichos objetos se destruyen → al pintar c2 los objetos r1 y r2 ya no existen y el destructor de Provincias ha liberado la memoria de nombre y capital.

r3 se ve bien porque no se destruye hasta ejecutar el delete

borramos la lista, pero no liberamos la memoria de los elementos de la lista que son punteros (si recuperar() lo ejecuto varias veces para un mismo objeto deberíamos liberar la memoria así: for(int i=0; i<cont;i++) delete lista[i]; //libera memoria objeto i-esimo delete [] lista; //borra la lista actual

Si se quiere guardar una estructura aún más compleja se hace necesario establecer un orden.

Si la clase Provincia es la que no tiene memoria dinámica (ni métodos para guardar y recuperar)

```
class ComunidadAutonoma {
    Provincias **lista;
    int nmax, cont;
public:
    ComunidadAutonoma(int _nmax=1) { //tamaño por defecto 1
        nmax = _nmax; //numero maximo de provincias
        cont = 0; //numero de provincias actuales
        lista = new Provincias*[nmax];
    }
    void add(Provincias &c) { //& indica paso por referencia
        if (cont==nmax) {
            Provincias **l=new Provincias*[nmax+3]; //crece de 3 en 3
            for(int i=0; i<nmax; i++) l[i]=lista[i];
            delete [] lista;
            lista=l;
            nmax+=3;
        }
        lista[cont++]=&c; //& indica direccion de memoria objeto c
    }
    bool almacenar(fstream &f) {
        f.write((char *)&nmax, sizeof(int)); //guardo nmax
        f.write((char *)&cont, sizeof(int)); //guardo cont
        for (int i=0; i<cont; i++)
            f.write( (char*)lista[i], sizeof( Provincias ) );
        //lista[i] > almacenar(f); //llamo a almacenar de Provincias
        return(f.good());
    }
    bool recuperar(fstream &f){
        delete [] lista; //borra la lista actual
        f.read((char *)&nmax, sizeof(int));
        f.read((char *)&cont, sizeof(int));
        lista = new Provincias*[nmax]; //reseva memoria para lista
        for (int i=0; i<cont; i++){
            Provincias *tmp= new Provincias(" ", " ", 0); //crea un objeto
            f.read( (char*)tmp, sizeof( Provincias ) );
            //tmp > recuperar(f);
            lista[i]=tmp;
        }
        return(f.good());
    }
    void pinta() {
        for (int i=0; i<cont; i++)
            lista[i]->pinta();
    }
};
```

```
c1:
Alava Victoria 300
Guipuzcoa San Sebastian 250
Vizcaya Bilbao 800

c2:
Vizcaya Bilbao 800
Guipuzcoa San Sebastian 250
Alava Victoria 300

Comunidad 1:
Alava Victoria 300
Guipuzcoa San Sebastian 250
Vizcaya Bilbao 800

Comunidad 2:
Vizcaya Bilbao 800
Guipuzcoa San Sebastian 250
Alava Victoria 300

Hay 2 Comunidades
```

```
class Provincias {
    char nombre[21]; char capital[21]; int censo;
public:
    void pinta() { cout << nombre << " " << capital
                    << " " << censo << endl; }
    Provincias( char *nom, char *cap, int cen ) {
        strcpy( nombre, nom );
        strcpy( capital, cap );
        censo = cen;
    }
};

int main(int argc, char *argv[]) {
    fstream archivo;
    ComunidadAutonoma c1(3), c2(2), c; //c(1)
    Provincias *r3; //puntero a Provincias
    {
        Provincias r1("Alava", "Victoria", 300);
        Provincias r2("Guipuzcoa", "San Sebastian", 250);
        r3=new Provincias("Vizcaya", "Bilbao", 800);
        c1.add(r1); c1.add(r2); c1.add(*r3);
        c2.add(*r3); c2.add(r2); c2.add(r1);
        cout << "c1:\n"; c1.pinta();
        cout << "nc2:\n"; c2.pinta();
        archivo.open( "Prueba.bin",
                    ios::out | ios::trunc | ios::binary );
        c1.almacenar( archivo ); //c1 salvado en fichero
        archivo.close();
        archivo.open( "Prueba.bin",
                    ios::out | ios::app | ios::binary );
        c2.almacenar( archivo ); //c2 salvado en fichero
        archivo.close();
    }
    //las variables locales al bloque se destruyen
    cout << "nc2:\n"; c2.pinta(); //r1 y r2 ya no existen
    archivo.open( "Prueba.bin", ios::in | ios::binary );
    c.recuperar( archivo );
    int i=0;
    while (!archivo.eof()) {
        cout << "nComunidad " << ++i << ":\n";
        c.pinta();
        c.recuperar( archivo );
    }
    cout << "nHay " << i << " Comunidades\n";
    archivo.clear(); archivo.close(); delete r3;
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Las objetos **r1** y **r2** son locales al bloque, con lo que al salir del bloque dichos objetos se destruyen → al pintar **c2** los objetos **r1** y **r2** ya no existen, el compilador marca esa zona de memoria como libre, pero mientras ‘tengamos la suerte’ de que el sistema no reserve esa zona de memoria para otra cosa, la información que hay allí no se pierde y por pantalla sale todo bien.

r3 ok porque no se destruye hasta ejecutar el **delete** borramos la lista, pero no liberamos la memoria de los elementos de la lista que son punteros (si **recuperar()** lo ejecuto varias veces para un mismo objeto deberíamos liberar la memoria así:

```
for(int i=0; i<cont; i++)
    delete lista[i]; //libera memoria objeto i-esimo
delete [] lista; //borra la lista actual
```

Versión con herencia: hay que saber el **tipo** de lo guardado, en la clase base los métodos almacenar y recuperar deben ser virtuales (para que haya polimorfismo) y en la clase hija dichos métodos deben internamente llamar a los de su clase padre siguiendo el mismo esquema utilizado con el operator=

```
class Provincias {
    char *nombre;
    char *capital;
    int censo;
protected:
    int tipo;
public:
    virtual void pinta() { cout << nombre << " " << capital
                        << " " << censo << endl; }
    Provincias( char *nom, char *cap, int cen ) { //constructor
        nombre=new char[strlen(nom)+1]; strcpy( nombre, nom );
        capital=new char[strlen(cap)+1];  strcpy( capital, cap );
        censo = cen;
        tipo=1;
    }
    virtual ~ Provincias () { delete [] nombre; delete [] capital; }
    virtual bool almacenar( fstream &f ){
        int n1=strlen(nombre)+1, n2=strlen(capital)+1;
        f.write((char *)(&tipo),sizeof(int));
        f.write((char *)(&n1),sizeof(int)); f.write((char *) (nombre),n1);
        f.write((char *)(&n2),sizeof(int)); f.write((char *) (capital),n2);
        f.write((char *)(&censo),sizeof(int)); //guardo censo
        return(f.good()); //true si se ha escrito bien
    }
    virtual bool recuperar(fstream &f){
        int n1, n2;
        f.read((char *)(&tipo),sizeof(int));
        f.read((char *)(&n1),sizeof(int)); //recupero tamaño
        delete [] nombre; delete [] capital; //libero memoria
        nombre=new char [n1];  f.read((char *) (nombre),n1);
        f.read((char *)(&n2),sizeof(int)); //recupero tamaño
        capital=new char [n2];  f.read((char *) (capital),n2);
        f.read((char *)(&censo),sizeof(int));
        return(f.good());
    }
};

class ProvinciaPlus:public Provincias {
    int nciudades;
    float ratio;
public:
    ProvinciaPlus ( char *nom, char *cap, int cen, int nc, float r )
        :Provincias( nom, cap, cen ) , nciudades(nc) { ratio=r; tipo=2; }
    bool almacenar( fstream &f ) {
        Provincias::almacenar(f); //mismo esquema que operator=
        f.write( (char *)(&nciudades), sizeof( int ) );
        f.write( (char *)(&ratio), sizeof( float ) );
        return(f.good());
    }
    bool recuperar( fstream &f ){
        Provincias::recuperar(f); //mismo esquema que operator=
        f.read( (char *)(&nciudades), sizeof( int ) );
        f.read( (char *)(&ratio), sizeof( float ) );
        return(f.good());
    }
    void pinta() {
        Provincias::pinta();
        cout << nciudades << " " << ratio << endl;
    }
};
```

```
class ComunidadAutonoma {
    Provincias **lista;
    int nmax,cont;
public:
    ... //el único que cambia es recuperar (lo amarillo)
    bool recuperar(fstream &f){
        delete [] lista; //borra la lista actual
        cout << f.good() << f.eof() << endl;
        f.read((char *)(&nmax),sizeof(int));
        cout << f.good() << f.eof() << endl;
        f.read((char *)(&cont),sizeof(int));
        lista = new Provincias*[nmax];
        int tipo=0;
        for (int i=0;i<cont;i++){
            //long plectura=f.tellg();
            f.read( (char *)(&tipo), sizeof( int ) );
            f.seekg(-(int)sizeof( int ), ios::cur);
            //f.seekg(plectura);
            if (tipo == 1) lista[i] = new Provincias(" ","",0);
            else if (tipo == 2)
                lista[i] = new ProvinciaPlus(" ","",0,0,0);
            else return 0; //VIP es que se ha llegado a EOF...
            lista[i]->recuperar(f);
        }
        return(f.good());
    }
    ...
};

int main(int argc, char *argv[]) {
    fstream archivo;
    ComunidadAutonoma c1(3), c2(2), c; //c(1)
    Provincias *r3; //puntero a Provincias
    { ProvinciaPlus r1("Alava","Victoria",300,50,3.5);
      Provincias r2("Guipuzcoa","San Sebastian",250);
      r3=new Provincias("Vizcaya","Bilbao",800);
      c1.add(r1); c1.add(r2); c1.add(*r3);
      c2.add(*r3); c2.add(r2); c2.add(r1);
      cout << "c1:\n"; c1.pinta();
      cout << "\nc2:\n"; c2.pinta();
      archivo.open( "pru.bin", ios::out | ios::trunc | ios::binary );
      c1.almacenar( archivo ); //c1 salvado en fichero
      archivo.close();
      archivo.open( "pru.bin", ios::out | ios::app | ios::binary );
      c2.almacenar( archivo ); //c2 salvado en fichero
      archivo.close();
    } //las variables locales al bloque se destruyen
    cout << "\nc2:\n"; c2.pinta(); //r1 y r2 ya no existen
    archivo.open( "pru.bin", ios::in | ios::binary );
    c.recuperar( archivo );
    int i=0;
    while (!archivo.eof()) {
        cout << "\nComunidad " << ++i << ":\n";
        c.pinta();
        c.recuperar( archivo );
    }
    cout << "\nHay " << i << " Comunidades\n";
    archivo.clear();  archivo.close(); delete r3;
    system("PAUSE");  return EXIT_SUCCESS;
}
```


Versión con herencia: hay que saber el **tipo** de lo guardado, en la clase base los métodos almacenar y recuperar deben ser virtuales (para que haya polimorfismo) y en la clase hija dichos métodos deben internamente llamar a los de su clase padre siguiendo el mismo esquema utilizado con el operador=

```
class Provincias {
    char *nombre; char *capital; int censo;
protected: int tipo;
public:
    virtual void pinta();
    Provincias( char *nom, char *cap, int cen ) { ... tipo=1; }
    virtual ~ Provincias () { delete [] nombre; delete [] capital; }
    virtual bool almacenar( fstream &f ){
        int n1=strlen(nombre)+1, n2=strlen(capital)+1;
        f.write((char *)&tipo,sizeof(int));
        f.write((char *)&n1,sizeof(int)); f.write((char *)nombre,n1);
        ...
    }
    virtual bool recuperar(fstream &f) {
        int n1, n2;
        f.read((char *)&tipo,sizeof(int));
        f.read((char *)&n1,sizeof(int)); //recupero tamaño
        ...
    }
};

class ProvinciaPlus:public Provincias {
    int nciudades; float ratio;
public:
    ProvinciaPlus ( char *nom, char *cap, int cen, int nc, float r )
        :Provincias( nom, cap, cen ) , nciudades(nc) { ratio=r; tipo=2; }
    bool almacenar( fstream &f ) {
        Provincias::almacenar(f);
        f.write( (char *)&nciudades, sizeof( int ) );
        f.write( (char *)&ratio, sizeof( float ) );
        return(f.good());
    }
    bool recuperar( fstream &f ){
        Provincias::recuperar(f);
        f.read( (char *)&nciudades, sizeof( int ) );
        f.read( (char *)&ratio, sizeof( float ) );
        return(f.good());
    }
    void pinta();
};
```

Lo gris es para comprender el porque del **return 0; VIP**. En la 2ª iteración del while del main estamos justo al final del fichero, invocamos **c.recuperar()** y al ejecutar **f.read((char *)&nmax,...);** alcanzamos EOF con lo que todas las lecturas read posteriores no se ejecutan → **tipo=0** y no se crea ningún objeto **lista[i]**. Si quitamos la línea **VIP** Se ejecutaría **lista[i]->recuperar(f);** y daría error en tiempo de ejecución ya que **lista[i]** no apuntan a ningún objeto. Quita esa línea y compruebe

```
c1:
Alava Victoria 300 50 3.5
Guipuzcoa San Sebastian 250
Vizcaya Bilbao 800
```

```
c2:
Vizcaya Bilbao 800
Guipuzcoa San Sebastian 250
Alava Victoria 300 50 3.5
```

```
c2:
Vizcaya Bilbao 800
8%R 250
Ç$R P%R 300
1 0 1 0
Comunidad 1:
Alava Victoria 300 50 3.5
Guipuzcoa San Sebastian 250
Vizcaya Bilbao 800
1 0 1 0
Comunidad 2:
Vizcaya Bilbao 800
Guipuzcoa San Sebastian 250
Alava Victoria 300 50 3.5
1 0 0 1
Hay 2 Comunidades
```

```
class ComunidadAutonoma {
    Provincias **lista;
    int nmax,cont;
public:
    ... //el único que cambia es recuperar (lo amarillo)
    bool recuperar(fstream &f){
        delete [] lista; //borra la lista actual
        cout << f.good() << f.eof() << endl;
        f.read((char *)&nmax,sizeof(int));
        cout << f.good() << f.eof() << endl;
        f.read((char *)&cont,sizeof(int));
        lista = new Provincias*[nmax];
        int tipo=0;
        for (int i=0;i<cont;i++){
            //long plectura=f.tellg();
            f.read( (char *)&tipo, sizeof( int ) );
            f.seekg(-(int)sizeof( int ), ios::cur);
            //f.seekg(plectura);
            if (tipo == 1) lista[i] = new Provincias(" "," ", 0);
            else if (tipo == 2)
                lista[i] = new ProvinciaPlus(" "," ",0,0,0);
            else return 0; //VIP es que se ha llegado a EOF...
            lista[i]->recuperar(f);
        }
        return(f.good());
    }
    ...
};

int main(int argc, char *argv[]) {
    fstream archivo;
    ComunidadAutonoma c1(3), c2(2), c; //c(1)
    Provincias *r3; //puntero a Provincias
    { ProvinciaPlus r1("Alava","Victoria",300,50,3.5);
      Provincias r2("Guipuzcoa","San Sebastian",250);
      r3=new Provincias("Vizcaya","Bilbao",800);
      c1.add(r1); c1.add(r2); c1.add(*r3);
      c2.add(*r3); c2.add(r2); c2.add(r1);
      cout << "c1:\n"; c1.pinta();
      cout << "\nc2:\n"; c2.pinta();
      archivo.open( "pru.bin", ios::out | ios::trunc | ios::binary );
      c1.almacenar( archivo ); //c1 salvado en fichero
      archivo.close();
      archivo.open( "pru.bin", ios::out | ios::app | ios::binary );
      c2.almacenar( archivo ); //c2 salvado en fichero
      archivo.close();
    } //las variables locales al bloque se destruyen
    cout << "\nc2:\n"; c2.pinta(); //r1 y r2 ya no existen
    archivo.open( "pru.bin", ios::in | ios::binary );
    c.recuperar( archivo );
    int i=0;
    while (!archivo.eof()) {
        cout << "\nComunidad " << ++i << ":\n";
        c.pinta();
        c.recuperar( archivo );
    }
    cout << "\nHay " << i << " Comunidades\n";
    archivo.clear(); archivo.close(); delete r3;
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Versión con herencia: hay que saber el **tipo** de lo guardado, en la clase base los métodos almacenar y recuperar deben ser virtuales (para que haya polimorfismo) y en la clase hija dichos métodos deben internamente llamar a los de su clase padre siguiendo el mismo esquema utilizado con el operator=

```
class Provincias {
    char *nombre;
    char *capital;
    int censo;
//protected:
//    int tipo;
public:
    virtual void pinta() { cout << nombre << " " << capital
                        << " " << censo << endl; }
    Provincias( char *nom, char *cap, int cen ) { //constructor
        nombre=new char[strlen(nom)+1]; strcpy( nombre, nom );
        capital=new char[strlen(cap)+1];  strcpy( capital, cap );
        censo = cen;
        //tipo=1;
    }
    virtual ~ Provincias () { delete [] nombre; delete [] capital; }
    virtual bool almacenar( fstream &f ){
        int n1=strlen(nombre)+1, n2=strlen(capital)+1;
        //f.write((char *)(&tipo),sizeof(int));
        f.write((char *)(&n1),sizeof(int)); f.write((char *)(&nombre),n1);
        f.write((char *)(&n2),sizeof(int)); f.write((char *)(&capital),n2);
        f.write((char *)(&censo),sizeof(int)); //guardo censo
        return(f.good()); //true si se ha escrito bien
    }
    virtual bool recuperar(fstream &f){
        int n1, n2;
        //f.read((char *)(&tipo),sizeof(int));
        f.read((char *)(&n1),sizeof(int)); //recupero tamaño
        delete [] nombre; delete [] capital; //libero memoria
        nombre=new char [n1];  f.read((char *)(&nombre),n1);
        f.read((char *)(&n2),sizeof(int)); //recupero tamaño
        capital=new char [n2];  f.read((char *)(&capital),n2);
        f.read((char *)(&censo),sizeof(int));
        return(f.good());
    }
    virtual bool almacenartipo( fstream &f ) {
        char tipo[200];
        strcpy(tipo, typeid(*this).name());
        int n1=strlen(tipo)+1;
        f.write((char *)(&n1),sizeof(int)); //guardo longitud del tipo
        f.write((char *)(&tipo),n1); //guardo el nombre del tipo
        return(f.good()); //true si se ha escrito bien
    }
    virtual void mostrartipo() {
        cout << typeid(*this).name() << endl;
    }
};
```

En vez de utilizar un atributo para codificar el tipo de lo guardado (implica crear un atributo adicional en la clase), una alternativa es guardar el nombre de la clase usando **typeid(objeto).name()** en un método **almacenartipo()** en la clase Base que debe ser virtual para que haya polimorfismo.

La clase que contenga objetos Base y Derivados debe almacenar el tipo en su método almacenar y recuperar la información del tipo en su método recuperar().

```
class ProvinciaPlus:public Provincias {
    int nciudades;
    float ratio;
public:
    ProvinciaPlus( char *n, char *c, int cen, int nc, float r )
        :Provincias( n, c, cen ) , nciudades(nc) { ratio=r; }
//tipo=2;
    bool almacenar( fstream &f ) {
        Provincias::almacenar(f); //esquema operator=
        f.write( (char *)(&nciudades), sizeof( int ) );
        f.write( (char *)(&ratio), sizeof( float ) );
        return(f.good());
    }
    bool recuperar( fstream &f ){
        Provincias::recuperar(f); //esquema operator=
        f.read( (char *)(&nciudades), sizeof( int ) );
        f.read( (char *)(&ratio), sizeof( float ) );
        return(f.good());
    }
    void pinta();
};

class ComunidadAutonoma {
    Provincias **lista;
    int nmax,cont;
public:
    ... //los que cambian son almacenar() y recuperar()
    bool almacenar(fstream &f) {
        f.write((char *)(&nmax),sizeof(int)); //guardo nmax
        f.write((char *)(&cont),sizeof(int)); //guardo cont
        for (int i=0;i<cont;i++) {
            cout << i << ": ";
            lista[i]->mostrartipo(); //muestro nombre del tipo
            lista[i]->almacenartipo(f); //almaceno nombre tipo
            lista[i]->almacenar(f);
        }
        return(f.good());
    }
    bool recuperar(fstream &f){
        delete [] lista; //borra la lista actual
        cout << f.good() << f.eof() << endl;
        f.read((char *)(&nmax),sizeof(int));
        cout << f.good() << f.eof() << endl;
        f.read((char *)(&cont),sizeof(int));
        lista = new Provincias*[nmax];
        char tipo[200]; int n1;
        for (int i=0;i<cont;i++){
            f.read((char *)(&n1),sizeof(int));
            f.read((char *)(&tipo),n1); //recupero nombre del tipo
            if (strcmp(tipo,typeid(Provincias).name())== 0)
                lista[i] = new Provincias(" ","",0);
            else if (strcmp(tipo,typeid(ProvinciaPlus).name())== 0)
                lista[i] = new ProvinciaPlus(" ","",0,0,0);
            else return 0; //VIP es que se ha llegado a EOF...
            lista[i]->recuperar(f);
        }
        return(f.good());
    }
    ...
};
```


Excepciones en los ficheros

ios:: exceptions

La mascara de excepciones esta compuesta por flags (bits) que representan si se emitirá una excepción en el caso de llegar a uno de dichos estados:

- **badbit** (critical error in stream buffer)
- **eofbit** (End-Of-File reached while extracting)
- **failbit** (failure extracting from stream)
- **goodbit** (no error condition, represents the absence of the above bits)

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

int main () {
    ifstream file;
    file.exceptions ( ifstream::eofbit |
ifstream::failbit | ifstream::badbit );
    try {
        file.open ("test.txt");
        while (!file.eof()) file.get();
    }
    catch (ifstream::failure e) {
        cout << "Exception opening/reading file";
    }

    file.close();

    return 0;
}
```

bool operator ! () (similar a good y fail)

si los flags de excepción están puestos devuelve el estado del stream

```
int main () {
    ifstream is;
    is.open ("test.txt");
    if (!is)
        cerr << "Error abriendo 'test.txt'\n";
    return 0;
}
```