

CAPÍTULO 4

©F.J.Ceballos/RA-MA

ENTRADA Y SALIDA ESTÁNDAR

Cuando se ejecuta un programa, se suceden fundamentalmente tres tareas: entrada de los datos, proceso de los mismos y salida o presentación de los resultados.



La tarea de entrada obtiene los datos necesarios para el programa de algún medio externo (por ejemplo, del teclado o de un fichero en disco) y los almacena en la memoria del ordenador para que sean procesados; un medio utilizado frecuentemente es el teclado. Por ejemplo:

```
cin >> a >> b; // leer a y b desde el teclado
```

El proceso de los datos dará lugar a unos resultados que serán almacenados temporalmente en memoria. Por ejemplo:

```
c = a + b; // sumar a y b; el resultado se almacena en c
```

La tarea de salida envía los resultados obtenidos a otro lugar; por ejemplo, los visualiza en el monitor, los escribe por la impresora o los guarda en un fichero en disco. La operación de salida no borra los datos de la memoria ni cambia la forma en la que están almacenados. Simplemente hace una copia de los mismos y la envía al lugar especificado; por ejemplo a la pantalla:

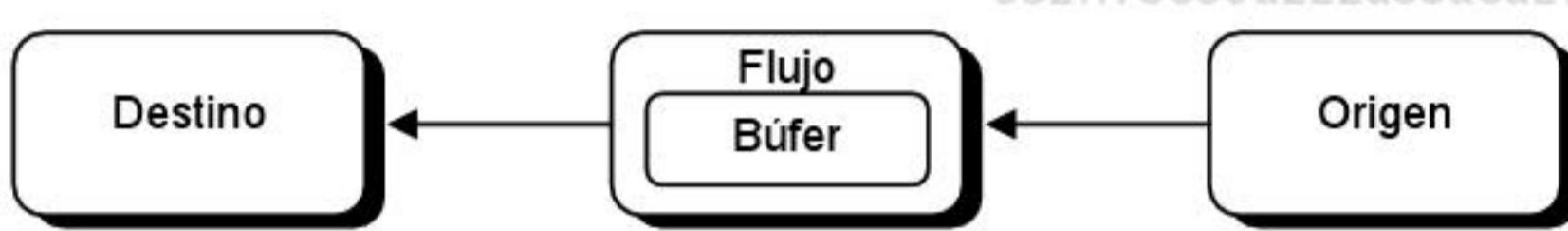
```
cout << c << endl; // mostrar el resultado c en la pantalla
```

En este capítulo estudiaremos los elementos de la biblioteca de C++ que permiten realizar operaciones de entrada y de salida (E/S) sobre los dispositivos

estándar del ordenador; esto es, cómo introducir datos desde el teclado y cómo visualizar datos en la pantalla.

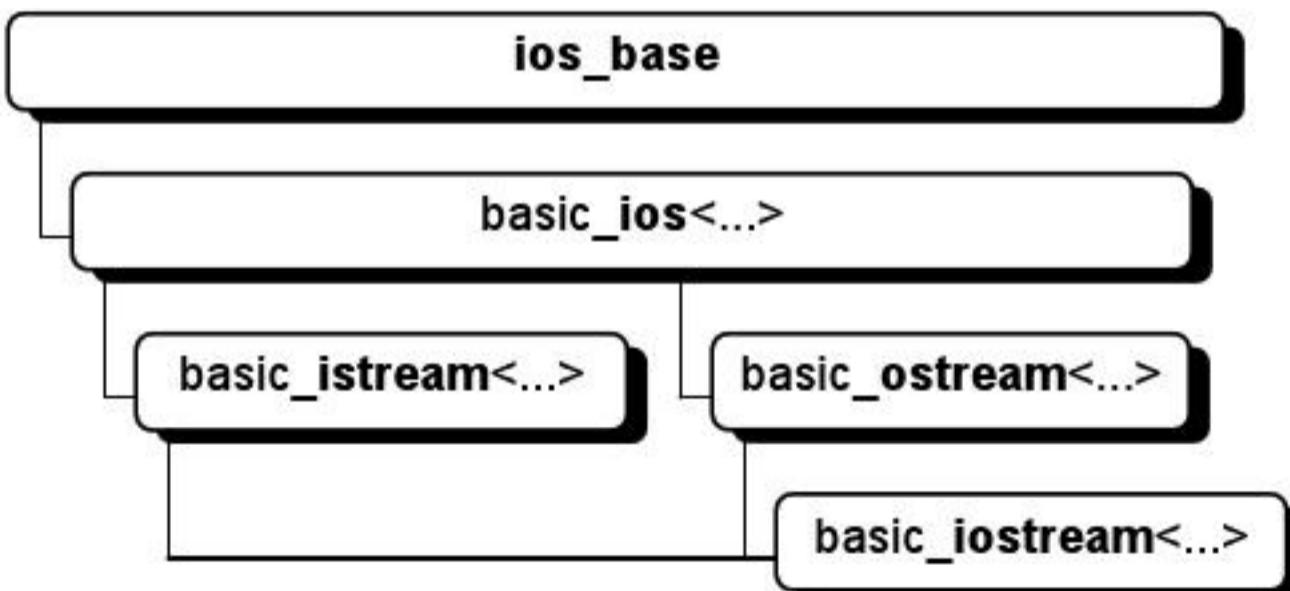
ENTRADA Y SALIDA

Frecuentemente un programa necesitará obtener información desde un origen o enviar información a un destino. Por ejemplo, obtener información desde el teclado, o bien enviar información a la pantalla. La comunicación entre el origen de cierta información y el destino se realiza mediante un *flujo* de información (en inglés *stream*).



Un *flujo* es un objeto que hace de intermediario entre el programa y el origen o el destino de la información. Esto es, el programa leerá o escribirá en el *flujo* sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que se traduce en una facilidad más a la hora de escribir programas.

La figura siguiente muestra las clases relacionadas con los flujos vinculados con la entrada y salida estándar. Todas las clases pertenecen al espacio std. La clase **ios** permite manipular operaciones generales de E/S. La clase **istream** se deriva de **ios** y permite manipular operaciones de entrada. La clase **ostream** se deriva de **ios** y permite manipular operaciones de salida. Y la clase **iostream** se deriva de **istream** y de **ostream** y permite manipular operaciones de E/S.



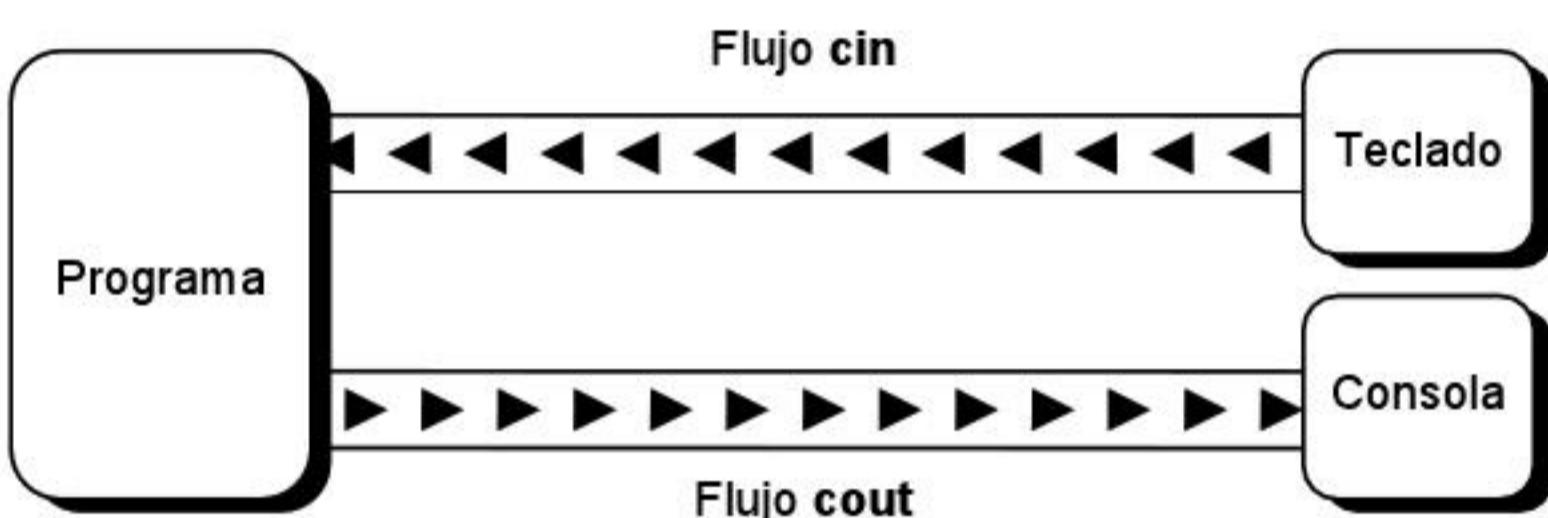
Exactamente, la clase **ios** se obtiene a partir de la plantilla **basic_ios<...>** particularizada para datos de tipo **char** (las plantillas serán estudiadas en un capítulo posterior):

```
typedef basic_ios<char> ios;
```

Análogamente, las clases **istream**, **ostream** e **iostream** se obtienen, respectivamente, a partir de las plantillas de clase **basic_istream**, **basic_ostream** y **basic_iostream**.

Cuando un programa C++ se ejecuta, se crean automáticamente tres flujos identificados por los objetos indicados a continuación:

- Un flujo desde la entrada estándar (el teclado): **cin**.
- Un flujo hacia la salida estándar (la consola): **cout**.
- Dos flujos hacia la salida estándar de error (la consola): **cerr** y **clog**.



El flujo **cin** es un objeto de la clase **istream** y los flujos **cout**, **cerr** y **clog** son objetos de la clase **ostream**.

Flujos de salida

Cuando un programa define un flujo de salida, por ejemplo el definido por el objeto **cout** de la clase **ostream**, el programa es el origen de ese flujo de bytes (es el que envía los bytes). Esta clase sobrecarga el operador **<<**, que recibe el nombre de operador de inserción, con el fin de ofrecer al programador una notación cómoda que le permita enviar al flujo de salida una secuencia de objetos en una única instrucción (más adelante, en otro capítulo, estudiaremos la sobrecarga de operadores, entendiendo por sobrecarga habilitar a un operador para operar con objetos para los que aún no está habilitado; por ejemplo, se puede habilitar el operador **+** para sumar números complejos). Los objetos **clog** y **cerr** se utilizan de igual manera que **cout**, aunque es habitual reservarlos para informes sobre operaciones realizadas o mensajes de error. Veamos un ejemplo:

```
cout << x; // escribe el valor de x
```

Si **x** es de tipo **int** con un valor 10, esta sentencia imprimirá 10. Análogamente, si **x** es de tipo **complex** con un valor (1.5, -2), esta sentencia imprimirá (1.5, -2). Esto sucederá así mientras **x** sea de un tipo para el que el operador **<<** esté sobrecargado, y lo está para todos los tipos primitivos y algunos derivados como las

cadenas de caracteres y el tipo **string**, y por ahora esto es todo lo que se necesita saber.

Cuando se escriben varias expresiones mediante una única sentencia, éstas se imprimirán en el orden esperado: de izquierda a derecha. Veamos un ejemplo:

```
cout << 'A' << ' ' << static_cast<int>('A') << '\n'; // escribe: A 65
```

Esta sentencia imprime un **char** (A), seguido de otro **char** (espacio en blanco), de un **int** (valor ASCII de A) y de otro **char** (cambio de línea).

El operador de inserción (<<), aunque esté sobrecargado, conserva su precedencia, que es suficientemente baja como para permitir expresiones aritméticas como operandos, sin tener que utilizar paréntesis. Por ejemplo:

```
cout << a+b*c << endl;
```

Esto quiere decir que se deben utilizar paréntesis para escribir expresiones que utilicen operadores con precedencia más baja que <<. Por ejemplo:

```
cout << (a&b|c) << endl;
```

Los argumentos para el operador de inserción pueden ser de cualquier tipo primitivo o derivado predefinido: **string**, **char[]**, **char**, **short**, **int**, **long**, **float**, **double**, **bool**, **unsigned int**, etc. Como ejemplo, el siguiente programa utiliza el operador << para escribir datos de varios tipos en el flujo cout.

```
#include <iostream>
3e2ff75c30d222a35aca2773f3e6e40d
ebrary #include <string>
using namespace std;

// Tipos de datos
int main()
{
    string sCadena = "Lenguaje C++";
    char cMatrizCars[] = "abc"; // matriz de caracteres
    int dato_int = 4;
    long dato_long = LONG_MIN; // mínimo valor long
    float dato_float = 3.40282347e+38F; // máximo valor float
    double dato_double = 3.1415926; // número PI
    bool dato_bool = true;
    cout << sCadena << endl;
    cout << cMatrizCars << endl;
    cout << dato_int << endl;
    cout << dato_long << endl;
    cout << dato_float << endl;
    cout << dato_double << endl;
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
    cout << boolalpha; // habilita que se escriba true o false
    cout << dato_bool << endl;
}
```

Los resultados que produce el programa anterior son los siguientes:

Lenguaje C++
abc
4
-2147483648
3.40282e+038
3.14159
true

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

En el ejemplo se puede observar también que la impresión de un objeto **string** hace que se imprima la cadena de caracteres que almacena.

Flujos de entrada

Cuando un programa define un flujo de entrada, por ejemplo el definido por el objeto **cin** de la clase **istream**, el programa es el destino de ese flujo de bytes (es el que recibe los bytes). Esta clase sobrecarga el operador **>>**, que recibe el nombre de operador de extracción, con el fin de ofrecer al programador una notación cómoda que le permita obtener del flujo de entrada una secuencia de objetos en una única instrucción (sobrecarga: vea el apartado *Flujos de salida*). Por ejemplo:

```
cin >> x; // obtiene el valor de x
```

Si *x* es de tipo **int** y se teclea un valor 10, esta sentencia asignará 10 a *x*. Análogamente, si *x* es de tipo **complex** y se teclea un valor (1.5, -2), esta sentencia asignará (1.5, -2) a *x*. Esto sucederá así mientras *x* sea de un tipo para el que el operador **>>** esté sobrecargado, y lo está para todos los tipos primitivos y algunos derivados como las cadenas de caracteres, y por ahora esto es todo lo que se necesita saber.

El siguiente código asigna un valor real a *n* desde el origen vinculado con el flujo **cin** (entrada estándar):

```
double n = 0;
cout << "Valor = "; // escribe: Valor =
cin >> n;           // asigna a n 10.5
```

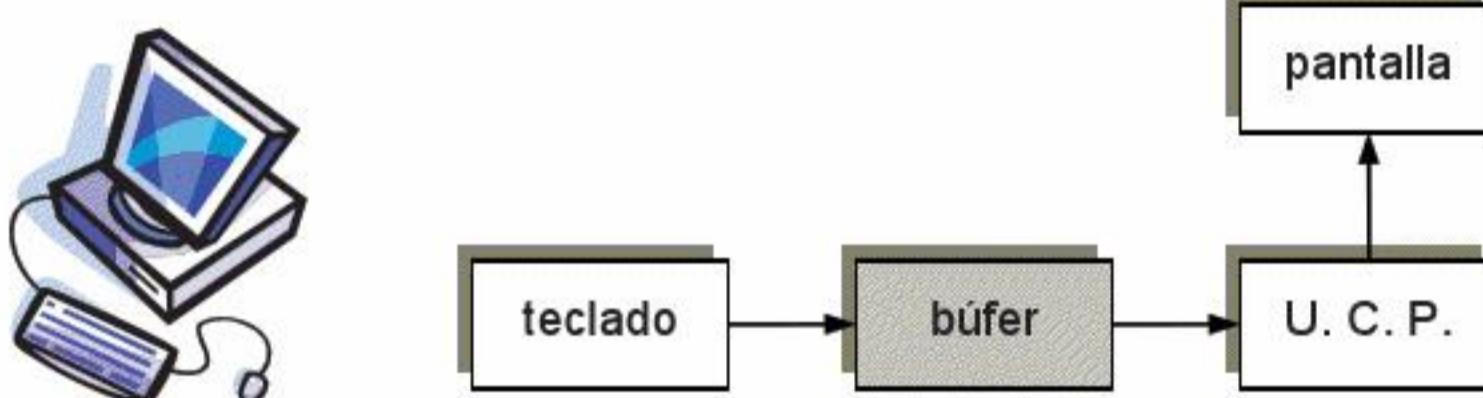
Cuando se leen varias variables mediante una única sentencia, a éstas les serán asignados los valores tecleados en el orden esperado: de izquierda a derecha. Por ejemplo:

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
int a;
float b;
char c;
cin >> a >> b >> c;
```

La última sentencia lee un **int**, seguido de un **float** y de un **char**. Estos datos serán introducidos, bien separados uno de otro pulsando la tecla *Entrar*, o bien separados por espacios en blanco, ya que **>>** salta los espacios en blanco.

Cuando un programa lee datos de la entrada estándar, como hace el operador **>>** sobre **cin**, la ejecución de dicho programa se detiene hasta que tecleemos los datos que hay que introducir y pulsemos la tecla *Entrar*. Los datos tecleados no son inmediatamente asignados a las variables especificadas y procesados; piense que si esto sucediera así no tendríamos opción a corregir un dato equivocado. Realmente, los datos se escriben en un búfer o memoria intermedia asociada con flujo de entrada y son enviados a la unidad central de proceso cuando se pulsa la tecla *Entrar* para ser asignados a las variables y ser procesados (**cout** y **clog** también envían los datos al flujo de salida a través de un búfer; **cerr** no utiliza búfer). Esos datos, según se escriben, son visualizados en el monitor con el fin de ver el proceso que estamos realizando.



Cuando se lee un dato, lo que hace el operador de extracción es obtener caracteres del búfer y convertirlos al formato de la variable que va a almacenarlos. En condiciones normales, la asignación a una variable finaliza cuando se llega a un separador. Este proceso se repite para cada una de las variables especificadas.

La ejecución del operador de extracción sobre **cin** finaliza cuando se han asignado valores a todas las variables o cuando se lee un carácter que no se corresponde con el formato de la variable. Por ejemplo, si para la misma sentencia anterior introducimos los datos:

5 tm 23.4 z [Entrar]

la operación de lectura se interrumpe porque la variable *b* espera un carácter válido para formar un real y ‘*t*’ no lo es. El resultado es que *a* vale 5 y no se asigna ningún valor ni a *b* ni a *c*. La ejecución continúa en la siguiente sentencia del programa, con los valores que tengan las variables *a*, *b* y *c*. El resultado final será inesperado porque no eran éstos los valores que deseábamos leer.

Los argumentos para el operador de extracción pueden ser de cualquier tipo primitivo o derivado predefinido: **string**, **char[]**, **char**, **short**, **int**, **long**, **float**, **double**, **bool**, **unsigned int**, etc. Como ejemplo, el siguiente programa utiliza el operador **>>** para obtener datos de varios tipos del flujo **cin**.

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

// Tipos de datos
int main()
{
    string sCadena = "Lenguaje C++";           3e2ff75c30d222a35aca2773f3e6e40d
    char cMatrizCars[80]; // matriz de caracteres      ebrary
    int dato_int = 0;
    long dato_long = 0;
    float dato_float = 0;
    double dato_double = 0;

    cout << "cadena (string): "; cin >> sCadena;
    cout << "cadena (char[]): "; cin >> cMatrizCars;
    cout << "int: "; cin >> dato_int;
    cout << "long: "; cin >> dato_long;
    cout << "float: "; cin >> dato_float;
    cout << "double: "; cin >> dato_double;

    cout << "\ndatos leídos:\n";
    cout << sCadena << endl;
    cout << cMatrizCars << endl;
    cout << dato_int << endl;
    cout << dato_long << endl;
    cout << dato_float << endl;
    cout << dato_double << endl;
}
```

Cuando un flujo se utiliza en una condición, su estado es analizado cada vez que ésta se ejecuta, siendo la condición verdadera solamente si el valor introducido es del tipo de la variable. Veamos un ejemplo:

```
int v = 0;
if (cin >> v)
    cout << v << endl;
else
    cout << "dato incorrecto\n";
```

En este ejemplo, la condición **cin >> v** fallará cuando los caracteres extraídos de la entrada estándar no puedan ser convertidos a un valor de tipo **int**.

ESTADO DE UN FLUJO

La clase **ios_base** es la clase base para todas las clases que definen flujos de E/S. Por lo tanto, sus métodos y atributos serán heredados por sus clases derivadas, lo que permite que un objeto de alguna de estas clases pueda invocar a cualquiera de los métodos heredados públicamente.

Cada flujo tiene un estado asociado con él (dado por un conjunto de bits), que puede ser analizado para manipular cualquier error que pueda ocurrir durante una operación de E/S. Los métodos para examinar el estado de un flujo están definidos en la plantilla **basic_ios** derivada de **ios_base** de la forma siguiente:

```
template< ... >
class basic_ios : public ios_base
{
    // ...

public:
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;
    // ...

};

typedef basic_ios<char>           ios;
```

ios::good() Devuelve **true** si todos los bits de error, incluido el de fin de fichero, están a 0. Esto es, la última operación de entrada ha tenido éxito. Por lo tanto, la próxima operación de entrada podría tener éxito; en otro caso, fallará.

ios::eof() Devuelve **true** si se encuentra el final de la entrada (*eof*). Esta acción se provoca también cuando por medio del método **clear** se pone el estado del flujo al valor **eofbit**.

ios::fail() Devuelve **true** si ocurre cualquier error, excepto *eof*. Esta acción se provoca también cuando por medio del método **clear** se pone el estado del flujo al valor **failbit** o **badbit**, indistintamente. Para saber si este error es recuperable (el flujo está sin corromper y no se han perdido caracteres), compruebe si **bad** devuelve **false**.

ios::bad() Devuelve **true** si ocurre un error irrecuperable (el flujo está corrompido). Esta acción se provoca también cuando por medio del método **clear** se pone el estado del flujo al valor **badbit**. En este estado deben abandonarse todas las operaciones de E/S.

Las constantes mencionadas en los párrafos anteriores, utilizadas para modificar el estado de un flujo, están definidas de la forma siguiente:

```
class ios_base
{
public:
    // ...
    static const iostate goodbit,
                badbit, // error irrecuperable
                eofbit, // fin de fichero
                failbit; // error recuperable
    // ...
};
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

ios::goodbit	No hay error (bits de error a 0).
ios::eofbit	Se encontró el final del fichero.
ios::failbit	Possible error recuperable, de formato o de conversión.
ios::badbit	Error irrecuperable.

Por ejemplo, el siguiente código intenta leer un valor **int**. Si el valor introducido para **x** no se corresponde con un entero, se mostrará el mensaje especificado.

```
int x = 0;
cin >> x; // operación de entrada
if (cin.fail())
    cout << "El dato no es correcto\n";
// ...
```

En el código anterior, si la última operación de entrada no tuvo éxito, esto es, si el método **fail** del objeto **cin** devolvió **true**, antes de realizar la próxima operación de entrada hay que poner a 0 todos los indicadores de error, de lo contrario no será posible realizarla, el código se ignorará. Para ello, utilice el método **clear** de la clase **ios** sin argumentos. Este método está definido como se indica a continuación (el valor por omisión del parámetro **f** es **goodbit**; los parámetros con valores por omisión serán estudiados en el capítulo *Más sobre funciones*):

```
void clear(iostate f = goodbit);
```

Si **f** es **goodbit**, se desactivan todos los indicadores de error. Si **f** es alguno de los valores **eofbit**, **failbit**, **badbit**, o una combinación de éstos utilizando el operador **or** (**|** o **bitor**), entonces se fija ese estado de error.

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

DESCARTAR CARACTERES DEL FLUJO DE ENTRADA

Los caracteres que no se extraen del búfer de entrada, por ejemplo, porque ocurrió un error debido a que nos equivocamos al teclearlos, pueden resultar indeseables. Para limpiar este búfer se puede utilizar el método **ignore** de la clase **istream**. Este método está definido así:

```
basic_istream& ignore(streamsize n = 1, int_type delim = traits_type::eof());
```

Este método extrae *n* caracteres (uno por omisión) y los descarta. La ejecución finaliza cuando ocurra algo de lo siguiente:

- Se hayan extraído *n* caracteres, si *n* != **numeric_limits<int>::max()**.
- Se detecte la marca de fin de fichero (*eof*, valor de *delim* por omisión).
- El carácter extraído coincide con el carácter indicado por *delim*.

El método **max** de la plantilla **numeric_limits**, declarada en el fichero de cabecera *limits*, retorna el valor más grande para el tipo especificado.

El siguiente ejemplo implementa una función para leer un dato **float** de forma segura; el dato será solicitado indefinidamente mientras el introducido no sea correcto. Así mismo, este ejemplo muestra cómo utilizar los métodos **clear** e **ignore**, además de otros, cuando ocurre un error recuperable después de un intento de leer datos del flujo de entrada. Obsérvese que, tras un error de entrada, primero, **clear** restablece a **goodbit** el estado del flujo y después, **ignore** elimina los datos que haya en el búfer de entrada hasta encontrar el carácter nueva línea introducido cuando se pulsó la tecla *Entrar*. Escriba la definición de la función en *utils.cpp* y su declaración en *utils.h* para permitir su reutilización en otros programas.

```
#include <iostream>
#include <limits> // necesario para numeric_limits<...>
using namespace std;

float LeerFloat()
{
    float x;
    while(true)
    {
        cin >> x;          // operación de entrada
        if (cin.good())
        {
            cin.ignore(); // eliminar \n del búfer (un carácter)
            return x;     // retornar el valor leído
        }
        else if (cin.fail()) // si la entrada no fue correcta...
    }
}
```

```
    cin.clear(); // poner a 0 todos los indicadores de error
    cin.ignore(numeric_limits<int>::max(), '\n');
}
}
}
```

La función anterior puede utilizarse así. Cree un nuevo proyecto y, además de *main.cpp*, añada al mismo *utils.cpp* (véase *Programa c++ formado por varios módulos* en el capítulo *Estructura de un programa*):

```
#include <iostream>
#include "utils.h" // contiene la declaración de LeerFloat
using namespace std;
int main()
{
    float dato_float = 0;

    cout << "float:           "; dato_float = LeerFloat();
    cout << dato_float << endl;
}
```

ENTRADA/SALIDA CON FORMATO

Los ejemplos realizados hasta ahora han mostrado sus resultados sin formato ninguno. Es decir, lo que hace el operador `<<` sobre el objeto `cout` es convertir los datos a imprimir en una secuencia de caracteres y mostrarlos sin más. Pero es evidente que en muchas ocasiones los resultados hay que mostrarlos según un formato y en un espacio determinado.

La plantilla de clase `basic_ios` derivada de la clase `ios_base` proporciona el control sobre los aspectos de cómo se produce la E/S. Para utilizar esta funcionalidad en una aplicación basta con incluir el fichero de cabecera *iostream*, a través del cual se incluye tanto la funcionalidad proporcionada por esta plantilla de clase como la de `basic_istream` y `basic_ostream` de donde se derivada. Por ejemplo:

```
int v = 165;

cout.setf(ios::hex, ios::basefield); // base 16 (hex)
cout.width(10); // campo de impresión de ancho 10
cout << v << endl; // escribe:      a5
```

Este ejemplo muestra el valor de la variable `v` en hexadecimal, indicado por los argumentos del método `setf` (el primer argumento especifica las opciones que se activan y el segundo, que es opcional, las que se desactivan) y ajustado a la derecha (por omisión) en un campo de ancho 10, indicado por el método `width`.

Para establecer u obtener la base, el tipo de alineación o el tipo de representación, **ios_base** define las siguientes constantes:

```
static const fmtflags basefield; // dec | oct | hex
static const fmtflags adjustfield; // left | right | internal
static const fmtflags floatfield; // scientific | fixed
```

Por ejemplo, para verificar si el tipo de alineación es a la izquierda, podemos escribir:

```
if ( (cout.flags() & ios::adjustfield) == ios::left ) ...
```

El método **flags** sobre un flujo devuelve un valor correspondiente al conjunto de opciones establecidas en dicho flujo.

No obstante, para facilitar las operaciones de E/S con formato, la biblioteca estándar ofrece un conjunto de indicadores de manipulación del estado de un flujo que cubren perfectamente la funcionalidad a la que nos hemos referido anteriormente. Estos indicadores son conocidos como *manipuladores*, son insertados directamente en la lista de expresiones de E/S y su finalidad es ejecutar determinadas acciones sobre las operaciones de E/S. Por ejemplo:

```
int v = 0;

cin >> oct >> v; // introducir un valor en base 8; por ejemplo 245
cout << hex << setw(10) << v << endl; // escribe: a5
cout << dec << setw(10) << v << endl; // escribe: 165
```

Este ejemplo solicita el valor de la variable **v** en octal, indicado por el manipulador **oct**, y lo muestra en hexadecimal y en decimal, indicado por los manipuladores **hex** y **dec**, y ajustado a la derecha (por omisión) en un campo de ancho 10, indicado por el manipulador **setw**. Seguramente, este ejemplo le resultará más fácil de entender que el anterior, en el que las operaciones se escribían en sentencias separadas, perdiendo en cierto modo las conexiones lógicas entre ellas.

Obsérvese que se han realizado dos operaciones justo antes de otra operación de salida. Pues bien, existe una gran variedad de operaciones que en ocasiones será interesante realizar justo antes o después de una operación de E/S. Los manipuladores que las permiten se encuentran en el espacio de nombres **std** y los hay sin parámetros (como **hex**) y con ellos (como **setw**). Muchos de ellos, localizados en los ficheros de cabecera *iostream* (incluye *istream* y *ostream*) e *iomanip*, se resumen a continuación:

boolalpha

Permitir mostrar los valores de tipo **bool** en formato alfabético; esta operación se desactiva con **noboolalpha**.

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

showbase	Permitir mostrar las constantes numéricas precedidas por un dígito distinto de 0, por 0 o por 0x, según se especifiquen en base 10, 8 ó 16, respectivamente; esta operación se desactiva con noshowbase .
showpoint	Forzar a que se muestre el punto decimal y los 0 no significativos en valores expresados en coma flotante; se desactiva con noshowpoint .
showpos	Mostrar el + para los valores positivos; esta operación se desactiva con noshowpos .
skipws	Saltar los espacios en blanco en la entrada (por omisión está activado); esta operación se desactiva con noskipws .
uppercase	Mostrar en mayúsculas los caracteres hexadecimales A-F y la E en la notación científica; esta operación se desactiva con nouppercase .
internal	Hacer que los caracteres de relleno se añadan después del signo o del indicador de base y antes del valor.
left	Alineación por la izquierda y relleno por la derecha.
right	Alineación por la derecha y relleno por la izquierda (establecido por omisión).
dec	Representación en decimal (base por omisión).
oct	Representación en octal.
hex	Representación en hexadecimal.
fixed	Activar el formato de coma flotante (dddd.dd).
scientific	Activar el formato en notación científica (d.ddddEdd).
endl	Escribir '\n' y vaciar el búfer del flujo.
ends	Escribir '\0'.
flush	Vaciar el búfer del flujo de salida.
ws	Saltar los espacios en blanco que preceden a un dato en la entrada. Mientras que skipws actúa sobre la entrada en general, ws se aplica sólo sobre el siguiente dato a leer, y se utiliza cuando skipws no está activado.
setiosflags(long)	Activar opciones como por ejemplo fixed , left , etc. (equivalente a setf). Se desactivan con resetiosflags (equivalente a unsetf).
setbase(int)	Establecer la base en la que se escribirán los enteros.
setfill(char)	Establecer como carácter de relleno el especificado.
setprecision(int)	Establecer el número de decimales para un valor real. La precisión por defecto es 6.
setw(int)	Establecer la anchura del campo donde se va a escribir un dato.

El siguiente ejemplo clarifica lo más significativo de lo expuesto hasta ahora.

```
#include <iostream>
#include <iomanip>
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```

using namespace std;

int main()
{
    int a = 12345;
    float b = 54.865F;

    cout << "      1      2" << endl;
    cout << "12345678901234567890" << endl;
    cout << "-----" << endl;
    cout << a << endl; // escribe 12345\n
    cout << '\n' << setw(10) << "abc"
        << setw(10) << "abcdef" << endl;

    cout << left; // se activa el ajuste a la izquierda
    cout << '\n' << setw(10) << "abc"
        << setw(10) << "abcdef" << endl;
    cout << endl; // avanza a la siguiente linea
    cout << right; // se vuelve al ajuste por la derecha

    // Se activa el formato de coma fija
    // con dos decimales
    cout << fixed << setprecision(2);
    cout << setw(15) << b << endl;
    cout << setw(15) << b/10 << endl;

    return 0;
}

```

Al ejecutar este programa se obtendrán los resultados mostrados a continuación. Observe que `\n` o `endl` avanzan al principio de la línea siguiente; si en este instante se envía a la salida otro `\n` o `endl`, estos dan lugar a una línea en blanco.

	1		2	
12345678901234567890				

12345				
	abc		abcdef	
abc		abcdef		
54.87				
5.49				

A continuación, damos una explicación de cada uno de los formatos empleados.

```
cout << a << endl; // escribe 12345\n
```

- escribe el entero *a*.
- **endl** avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << '\n' << setw(10) << "abc" << setw(10) << "abcdef" << endl;
```

- **\n** avanza a la línea siguiente.
- **setw(10)** para escribir la cadena “*abc*” sobre un ancho de 10 posiciones. La cadena se ajusta por defecto a la derecha.
- **setw(10)** para escribir la cadena “*abcdef*” sobre un ancho de 10 posiciones. La cadena se ajusta por defecto a la derecha.
- **endl** avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << left;
```

- activa el ajuste por la izquierda en el ancho que se establezca.

```
cout << '\n' << setw(10) << "abc" << setw(10) << "abcdef" << endl;
```

- igual que anteriormente pero con ajustes a la izquierda.

```
cout << endl;
```

- avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << right;
```

- activa el ajuste por la derecha en el ancho que se establezca.

```
cout << fixed << setprecision(2);
```

- activa el formato de coma fija con dos decimales.

```
cout << setw(15) << b << endl;
```

- **setw(15)** para escribir la cadena *b* sobre un ancho de 15 posiciones. El ajuste está establecido por la derecha. El valor de *b* se redondea a dos decimales.
- **endl** avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << setw(15) << b/10 << endl;
```

- análoga a la anterior. Observe cómo las dos últimas cantidades escritas quedan ajustadas por su parte entera y decimal.

Este otro ejemplo que se presenta a continuación ilustra cómo se utilizan los indicadores de formato a través de **setiosflags**. El resultado que se quiere obtener es el siguiente:

```
Madrid..... 5198.00  
Sevilla..... 3.21
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

Valencia.....	46.32
Cantabria.....	506.50
Barcelona.....	2002.38

Tanto los nombres de las provincias como los coeficientes asociados estarán almacenados en sendas matrices (las matrices las estudiaremos en un capítulo posterior).

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

int main()
{
    double coef[] = { 5198.0, 3.21, 46.32, 506.5, 2002.38 };
    string prov[] = { "Madrid", "Sevilla", "Valencia", "Cantabria",
                      "Barcelona" };
    // Salida de resultados alineados en columnas
    cout << setiosflags( ios::fixed ); // formato en coma flotante
    for ( int i = 0; i < sizeof( coef )/sizeof( double ); i++)
        cout << setiosflags( ios::left ) // justificación a la izda.
            << setw( 15 ) // ancho para las cadenas de caracteres
            << setfill( ' ' ) // carácter de relleno
            << prov[i] // escribe la provincia
            << resetiosflags( ios::left ) // suprime justificación
            << setw( 10 ) // ancho para las cantidades
            << setprecision( 2 ) // dos decimales
            << coef[i] << endl; // escribe cantidad y '\n'
}
```

La sentencia **for**, que será estudiada en un capítulo posterior, tiene como misión ejecutar la sentencia de salida tantas veces como elementos hemos almacenado en las matrices.

Se puede especificar como argumento de **setiosflags** varios indicadores de formato unidos por el operador | (*or*).

ENTRADA DE CARACTERES

El operador **>>** sobre **cin** está pensado para aceptar valores separados por espacios para variables de un tipo esperado. Cuando lo que se desea es leer caracteres como tales (incluidos los espacios) se utiliza el método **get** de **basic_istream**. Cada vez que se ejecute este método se leerá el siguiente carácter al último leído. Su sintaxis es así:

```
basic_istream& get(char_type& car);
```

El método **get** almacena en *car* el carácter leído; si el carácter leído coincide con el final del fichero, pone el estado del flujo al valor **eofbit**. Por ejemplo:

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

int main()
{
    char car = 0;
    cout << "Introducir un carácter: ";
    // Leer un carácter y almacenarlo en la variable car
    cin.get(car);
    cout << "Carácter: " << car
        << ", valor ASCII: " << static_cast<int>(car) << endl;
}
```

Ejecución del programa

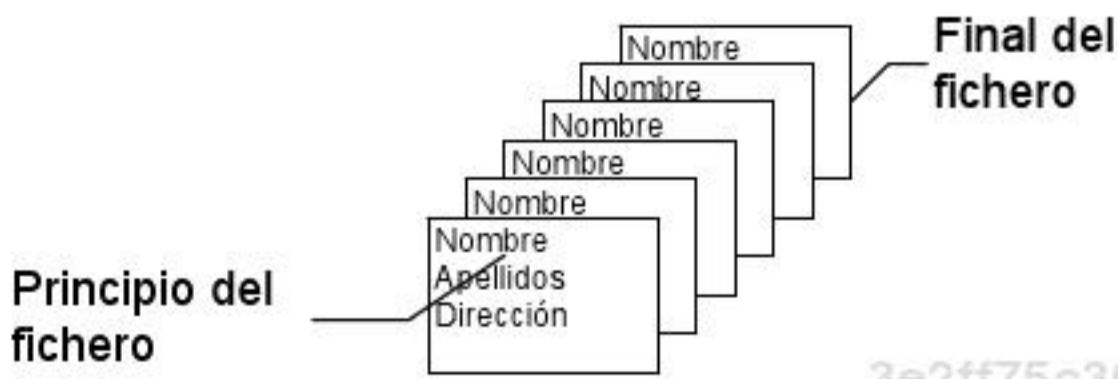
Introducir un carácter: a
Carácter: a, valor ASCII: 97

En el ejemplo anterior se puede observar que dependiendo del formato que se utilice para visualizar *car*, **char** o **int**, el resultado puede ser el propio carácter o su valor ASCII. Internamente, refiriéndonos a la memoria del ordenador, no hay más que un conjunto de 0 y 1 (*01100001*).

Suponiendo que el búfer asociado con la entrada estándar está vacío, cuando en el programa anterior se ejecute el método **get**, su ejecución se detendrá hasta que tecleemos un carácter y pulsemos la tecla *Entrar*. El carácter leído será almacenado en la variable *car*.

CARÁCTER FIN DE FICHERO

Desde el punto de vista del desarrollador de una aplicación, un dispositivo de entrada o de salida estándar es manipulado por el lenguaje C++ como si de un fichero de datos en el disco se tratara. Un fichero de datos no es más que una colección de información. Los datos que introducimos por el teclado son una colección de información y los datos que visualizamos en el monitor son también una colección de información.



Todo fichero tiene un principio y un final. ¿Cómo sabe un programa que está leyendo datos de un fichero, que se ha llegado al final del mismo y, por lo tanto, no hay más datos? Por una marca de fin de fichero. En el caso de un fichero grabado en un disco esa marca estará escrita al final del mismo. En el caso del teclado la información procede de lo que nosotros tecleamos, por lo tanto si nuestro programa requiere detectar la marca de fin de fichero, tendremos que teclearla cuando demos por finalizada la introducción de información. Esto se hace pulsando las teclas *Ctrl+D* en UNIX o *Ctrl+Z* en una aplicación de consola en Windows.

Ya que un fichero o un dispositivo siempre son manejados a través de un flujo, hablar del final del flujo es sinónimo de hablar del final del fichero. Por eso, de ahora en adelante, siempre que tengamos que realizar algún tipo de operación sobre un dispositivo o sobre un fichero, nos referiremos indistintamente a ellos o al flujo que los representa.

Cuando el operador `>>` sobre `cin` intenta leer un carácter fin de fichero, pone el estado del flujo al valor **eofbit** (ídem para `get`). También activa el indicador **failbit** puesto que la entrada no es válida. Utilice `clear` para desactivar los indicadores y permitir realizar nuevas lecturas. Veamos un ejemplo. El programa siguiente escribe la suma de los datos introducidos por el teclado. La entrada finalizará cuando se introduzcan *n* datos o se detecte el *eof* (*Ctrl+Z* | *Ctrl+D* seguido de la tecla *Entrar*).

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

int main()
{
    int dato, suma = 0, n = 10, i = 0;
    bool eof = false;

    cout << "Introducir n datos máximo. Finalizar con eof.\n\n";

    do
    {
        cout << "dato: ";
        cin >> dato;
        eof = cin.eof();
        if (!eof && cin.fail())
        {
            cerr << '\a' << "\ndato incorrecto\n";
            cin.clear(); // desactivar los bits de error
            cin.ignore(numeric_limits<int>::max(), '\n');
        }
        else if (cin.good())
        {
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
    suma += dato;
    i++;
}
while( !eof && !cin.bad() && i < n );
cout << "Suma: " << suma << endl;
}
```

En el capítulo *Sentencias de control* veremos las sentencias **while** y **do...while**. Ahora le basta con saber que ambas ejecutan su bloque de instrucciones (el encerrado entre { y }) mientras la condición escrita a continuación de la palabra reservada **while** sea verdad.

CARÁCTER \n

Cuando se están introduciendo datos a través del teclado y pulsamos la tecla *Entrar* se introduce también el carácter `\n`. Mientras que en la salida `\n` produce un *CR+LF* (*CR* es el ASCII 13 y *LF* es el ASCII 10), en la entrada se corresponde con un *LF*; esto es, una expresión C++ como '`\n`' == 10 daría como resultado `true`. En algunas ocasiones, este carácter `\n` puede tener efectos indeseables. Vamos a explicar esto con el siguiente ejemplo que, inicialmente, lee un número real:

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

int main()
{
    float precio = 0.0F;
    cout << "Precio: ";
    cin >> precio;
    cout << "Precio = " << precio << endl;
}
```

Cuando se ejecute el operador `>>` sobre `cin` del programa anterior, si tecleamos 1000 y pulsamos la tecla *Entrar*, antes de la lectura habrá en el búfer de entrada (búfer asociado con `cin`) la siguiente información:

y después de la lectura,

\n

ya que el carácter `\n` no es un carácter válido para un valor **float**; por lo tanto, aquí se interrumpe la lectura. Este carácter sobrante puede ocasionarnos problemas si a continuación se ejecuta otra sentencia de entrada que admita datos que sean caracteres como sucede con **get**. Por ejemplo, para ver este detalle vamos a modificar el programa anterior de la siguiente forma:

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

int main()
{
    float precio = 0.0F;
    char car = 0;

    cout << "Precio: " >> precio;
    cout << "Pulse <Entrar> para continuar ";
    cin.get(car);
    cout << "Precio = " << precio << endl;
}
```

Si ejecutamos este programa y tecleamos el dato 1000, se producirá el siguiente resultado:

```
Precio: 1000[Entrar]
Pulse <Entrar> para continuar Precio = 1000
```

A la vista del resultado, se observa que no se ha hecho una pausa. ¿Por qué? Porque el carácter sobrante nueva línea es un carácter válido para el método **get**, razón por la que **get** no necesita esperar a que introduzcamos un carácter para la variable **car**.

La solución al problema planteado es limpiar el búfer asociado con **cin** antes de ejecutarse **get**, utilizando el método **ignore**.

```
int main()
{
    float precio = 0.0F;
    char car = 0;

    cout << "Precio: " >> precio;
    cout << "Pulse <Entrar> para continuar ";
    cin.ignore(); // eliminar el '\n' sobrante
    cin.get(car);
    cout << "Precio = " << precio << endl;
}
```

ENTRADA DE CADENAS DE CARACTERES

El operador `>>` sobre `cin` está pensado para aceptar valores separados por espacios. Entonces, ¿cómo leemos una cadena de caracteres de tipo `string` que contenga espacios en blanco? Pues invocando a la función `getline`:

```
istream& getline (istream& is, string& str);
```

Esta función lee caracteres de la entrada estándar (representada por `cin`) hasta encontrar el carácter `'\n'` que se introduce al pulsar la tecla *Entrar*, incluido este carácter. El hecho de leer también el carácter `'\n'` puede presentar un problema cuando antes se haya leído otro dato de un tipo primitivo. Por ejemplo:

```
cout << "Introducir un real: ";
cin >> dDato;
cout << "Introducir una cadena de caracteres: ";
getline(cin, strDato);
// ...
```

En este ejemplo, `getline` lee el carácter `'\n'` que se introduce al pulsar la tecla *Entrar* después de introducir el valor real, asumiendo que esta es la cadena que queríamos leer (se trata de una cadena vacía), con lo que el usuario no podrá introducir la cadena solicitada. La solución es eliminar ese carácter `'\n'` sobrante antes de que sea solicitada la entrada de la cadena de caracteres:

```
// ...
cout << "Introducir un real: ";
cin >> dDato;
cin.ignore(); // eliminar el '\n' sobrante
cout << "Introducir una cadena de caracteres: ";
getline(cin, strDato);
// ...
```

EJERCICIOS RESUELTOS

1. Realizar un programa que dé como resultado los intereses producidos y el capital total acumulado de una cantidad c , invertida a un interés r durante t días.

La fórmula utilizada para el cálculo de los intereses es:

$$I = \frac{c * r * t}{360 * 100}$$

siendo:

I = Total de intereses producidos.

c = Capital.

r = Tasa de interés nominal en tanto por ciento.

t = Período de cálculo en días.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double c, intereses, capital;
float r;
int t;
```

- A continuación leemos los datos c , r y t .

```
cout << "Capital invertido      "; cin >> c;
cout << "\nA un % anual del    "; cin >> r;
cout << "\nDurante cuántos días  "; cin >> t;
```

- Conocidos los datos, realizamos los cálculos. Nos piden los intereses producidos y el capital acumulado. Los intereses producidos los obtenemos aplicando directamente la fórmula. El capital acumulado es el capital inicial más los intereses producidos.

```
intereses = c * r * t / (360 * 100);
capital = c + intereses;
```

- Finalmente, escribimos el resultado.

```
cout << fixed << setprecision(2);
cout << "Intereses producidos..." << setw(12) << intereses << endl;
cout << "Capital acumulado...." << setw(12) << capital << endl;
```

El programa completo se muestra a continuación.

```
// Capital e Intereses
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double c, intereses, capital;
    float r;
    int t;

    // Entrada de datos
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
cout << "Capital invertido      "; cin >> c;
cout << "\nA un % anual del      "; cin >> r;
cout << "\nDurante cuántos días    "; cin >> t;
cout << "\n\n";

// Cálculos
intereses = c * r * t / (360L * 100);
capital = c + intereses;

// Escribir resultados
cout << fixed << setprecision(2);
cout << "Intereses producidos..." << setw(12) << intereses << endl;
cout << "Capital acumulado....." << setw(12) << capital << endl;
}
```

Ejecución del programa:

Capital invertido 1000000
A un % anual del 8
Durante cuántos días 360

Intereses producidos... 80000.00
Capital acumulado..... 1080000.00

2. Realizar un programa que calcule las raíces de la ecuación:

$$ax^2 + bx + c = 0$$

teniendo en cuenta los siguientes casos:

- a) Si a es igual a 0 y b es igual a 0, imprimiremos un mensaje diciendo que la ecuación es degenerada.
- b) Si a es igual a 0 y b no es igual a 0, existe una raíz única con valor $-c / b$.
- c) En los demás casos, utilizaremos la fórmula siguiente:

$$x_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

La expresión $d = b^2 - 4ac$ se denomina discriminante.

- Si d es mayor o igual que 0 entonces hay dos raíces reales.
- Si d es menor que 0 entonces hay dos raíces complejas de la forma:

$$x + yj, x - yj$$

Indicar con literales apropiados los datos a introducir, así como los resultados obtenidos.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double a, b, c; // coeficientes de la ecuación
double d; // discriminante
double re, im; // parte real e imaginaria de la raíz
```

- A continuación leemos los datos *a*, *b* y *c*.

```
cout << "Coeficientes a, b y c de la ecuación: ";
cin >> a >> b >> c;
```

- Leídos los coeficientes, pasamos a calcular las raíces.

```
if (a == 0 && b == 0)
    cout << "La ecuación es degenerada\n";
else if (a == 0)
    cout << "La única raíz es: " << -c / b << endl;
else
{
    // Evaluar la fórmula. Cálculo de d, re e im
    if (d >= 0)
    {
        // Imprimir las raíces reales
    }
    else
    {
        // Imprimir las raíces complejas conjugadas
    }
}
```

- Cálculo de $\frac{-b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}$

$$\begin{array}{ccc} & \nearrow & \nearrow \\ & re & im \end{array}$$

```
re = -b / (2 * a);
d = b * b - 4 * a * c;
im = sqrt(fabs(d)) / (2 * a);
```

- Imprimir las raíces reales.

```
cout << "Raíces reales:\n";
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
cout << re + im << " y " << re - im << endl;
```

- Imprimir las raíces complejas conjugadas.

```
cout << "Raíces complejas:\n";
cout << re << " + " << fabs(im) << " j" << endl;
cout << re << " - " << fabs(im) << " j" << endl;
```

El programa completo se muestra a continuación.

```
// ecu2gra.cpp - Raíces de una ecuación de 2º grado
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
    double a, b, c; // coeficientes de la ecuación
    double d; // discriminante
    double re, im; // parte real e imaginaria de la raíz

    cout << "Coeficientes a, b y c de la ecuación: ";
    cin >> a >> b >> c;
    cout << '\n';

    cout << fixed << setprecision(2);
    if (a == 0 && b == 0)
        cout << "La ecuación es degenerada\n";
    else if (a == 0)
        cout << "La única raíz es: " << -c / b << endl;
    else
    {
        re = -b / (2 * a);
        d = b * b - 4 * a * c;
        im = sqrt(fabs(d)) / (2 * a);
        if (d >= 0)
        {
            cout << "Raíces reales:\n";
            cout << re + im << " y " << re - im << endl;
        }
        else
        {
            cout << "Raíces complejas:\n";
            cout << re << " + " << fabs(im) << " j" << endl;
            cout << re << " - " << fabs(im) << " j" << endl;
        }
    }
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

Ejecución del programa

Coeficientes a, b y c de la ecuación: 1 -2 3

Raíces complejas:

1.00 + 1.41 j

1.00 - 1.41 j

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

- 1) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    float a = 0.0F;
    a = 1/3;
    cout<< fixed << a << endl;
}
```

- a) 0.000000.
- b) 0.333333.
- c) 1.0.
- d) 0.33333.

- 2) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
int main( )
{
    using namespace std;
    float a = 10.0F, b;
    b = a/2;
    cout << b << endl;
}
```

- a) 5.000000.
- b) 5.
- c) 5.0.
- d) 5.00000.

- 3) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
```

```
#include <iomanip>
int main( )
{
    using namespace std;
    double a = 20.0, b;
    b = a/3;
    cout << fixed << b << endl;
```

- a) Error. El formato tiene que ser %lf.
- b) 6.66666.
- c) 6.666666.
- d) 6.666667.

4) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    int a = 10;
    cout << hex << uppercase << a << endl;
```

- a) 10.
- b) 0xA.
- c) A.
- d) a.

5) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    double a = 20.0, b;
    b = a/3;
    cout << fixed << setprecision(2);
    cout << setw(20) << b << endl;
```

- a) 6.666666 ajustado al margen izquierdo de la pantalla.
- b) 6.666666 ajustado a la derecha en un ancho de 20 posiciones.
- c) 6.67 ajustado a la derecha en un ancho de 22 posiciones.
- d) Ninguno de los anteriores.

- 6) ¿Cuál es el resultado del siguiente programa suponiendo que se teclea el valor 3.1416?

```
#include <iostream>
int main( )
{
    using namespace std;
    long a = 0;
    cin >> a;
    cout << a << endl;
}
```

- a) 3.1416.
- b) 3.
- c) 3.142.
- d) 0.0.

- 7) ¿Cuál es el resultado del siguiente programa suponiendo que se teclea el valor 3.1416?

```
#include <iostream>
int main( )
{
    using namespace std;
    long a = 0;
    double b = 0.0;
    cin >> a >> b;
    cout << b << endl;
}
```

- a) 3.1416.
- b) Error durante la ejecución.
- c) 0.1416.
- d) 0.0.

- 8) ¿Cuál es el resultado del siguiente programa suponiendo que se teclean los valores: 3.1416 x6A -3.1?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    double a = 0.0, b = 0.0, c = 0.0;
    bool r = true;
    cin >> a >> b >> c;
    r = cin.rdstate() == ios::goodbit;
    cout << boolalpha << r << endl;
}
```

- a) true.
b) false.
c) 1.
d) 0.
- 9) ¿Cuál es el resultado del siguiente programa suponiendo que el valor ASCII del 0 es 48 y que se teclea el valor 2?

```
#include <iostream>
int main( )
{
    using namespace std;
    char c = 0;
    cin.get(c);
    cout << c * 2 << endl;
}
```

- a) 100.
b) 96.
c) 4.
d) Ninguno de los anteriores.
- 10) ¿Cuál es el resultado que se ve en la pantalla cuando se ejecuta el siguiente programa suponiendo que se teclean los valores: 2 s?

```
#include <iostream>
int main( )
{
    using namespace std;
    int a = 0; char c = 0;
    cin >> a; cin.get(c);
    cout << a << ' ' << c << endl;
}
```

- a) 2 s.
b) 2.
c) s.
d) Ninguno de los anteriores.
2. Realizar un programa que calcule el volumen de una esfera, que viene dado por la fórmula:

$$V = \frac{4}{3} \pi r^3$$

3. Realizar un programa que pregunte el nombre, el año de nacimiento y el año actual, y dé como resultado:

Hola nombre, en el año 2030 cumplirás n años

4. Realizar un programa que evalúe el polinomio $p = 3x^5 - 5x^3 + 2x - 7$ y visualice el resultado con el siguiente formato:

Para $x = \text{valor}$, $3x^5 - 5x^3 + 2x - 7 = \text{resultado}$

5. Realizar el mismo programa anterior, pero empleando ahora coeficientes variables a , b y c . Piense ahora en objetos y escriba una clase *CEcuacion* con los atributos a , b y c y los métodos *establecerCoeficientes* y *valorEcuacion*, además de los constructores necesarios. La función **main** creará un objeto ecuación y visualizará su valor para unos coeficientes y un valor de x introducidos por el teclado.

6. Ejecute el siguiente programa, explique lo que ocurre y realice las modificaciones que sean necesarias para su correcto funcionamiento.

```
#include <iostream>
int main( )
{
    using namespace std;
    cout << "Introducir un carácter: "; cin.get(car);
    cout << car << endl;
    cout << "Introducir otro carácter: "; cin.get(car);
    cout << car << endl;
}
```

7. Indique qué resultado da el siguiente programa. A continuación ejecute el programa y compare los resultados.

```
#include <iostream>
int main( )
{
    using namespace std;
    char car1 = 'A', car2 = 65, car3 = 0;
    car3 = car1 + 'a' - 'A';
    cout << static_cast<int>(car3) << ' ' << car3 << endl;
    car3 = car2 + 32;
    cout << static_cast<int>(car3) << ' ' << car3 << endl;
}
```