CAPÍTULO 5

CLASES

Seguro que a estas alturas el término *clase* ya le es familiar. En los capítulos expuestos hasta ahora se han desarrollado aplicaciones sencillas, para introducirle en el diseño de clases y en el manejo de la biblioteca de clases de C++. Por lo tanto, ya habrá asimilado que un programa orientado a objetos sólo se compone de objetos y que un objeto es la concreción de una clase. Es hora pues de entrar con detalle en la programación orientada a objetos, la cual tiene un elemento básico: la *clase*.

DEFINICIÓN DE UNA CLASE

Una *clase* es un tipo definido por el usuario que describe los atributos y los métodos de los objetos que se crearán a partir de la misma. Los *atributos* definen el estado de un determinado objeto y los *métodos* son las operaciones que definen su comportamiento. Forman parte de estos métodos los *constructores*, que permiten iniciar un objeto, y los *destructores*, que permiten destruirlo. Los atributos y los métodos se denominan en general *miembros* de la clase.

Según hemos aprendido, la definición de una clase consta de dos partes: el *nombre de la clase* precedido por la palabra reservada **class** y el *cuerpo de la clase* encerrado entre llaves y seguido de un punto y coma. Esto es:

```
class nombre_clase
{
    cuerpo de la clase
};
```

El *cuerpo de la clase* en general consta de modificadores de acceso (**public**, **protected** y **private**), atributos, mensajes y métodos. Un método implícitamente define un mensaje (el nombre del método es el mensaje).

Por ejemplo, un círculo puede ser descrito por la posición (x, y) de su centro y por su *radio*. Hay varias cosas que nosotros podemos hacer con un círculo: calcular la longitud de la circunferencia, calcular el área del círculo, etc. Cada círculo es diferente (por ejemplo, tienen el centro o el radio diferente); pero visto como una *clase* de objetos, el círculo tiene propiedades intrínsecas que nosotros podemos agrupar en una definición. El siguiente ejemplo define la clase *Círculo*. Obsérvese cómo los atributos y los métodos forman el cuerpo de la clase.

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

class Circulo
{
  // miembros privados
  private:
   double x, y;      // coordenadas del centro
   double radio;     // radio del círculo

  // miembros protegidos
  protected:
    void msgEsNegativo()
    {
      cout << "El radio es negativo. Se convierte a positivo\n";
    }

  // miembros públicos
  public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r) // constructor
    {
      x = cx; y = cy;
      if (r < 0)
      {
        msgEsNegativo();
        r = -r;
      }
      radio = r;
    }

    double longCircunferencia()
    {
      return 2 * 3.1415926 * radio;
    }

    double areaCirculo()
    {
      return 3.1415926 * radio * radio;
    }
};
```

Este ejemplo define un nuevo tipo de datos, *Círculo*, que puede ser utilizado dentro de un programa fuente exactamente igual que cualquier otro tipo. Un objeto de la clase *Círculo* tendrá los atributos *x*, *y* y *radio*, los métodos *msgEsNegativo*, *longCircunferencia* y *areaCírculo* y dos constructores *Círculo*, uno sin parámetros y otro con ellos.

Atributos

Los atributos constituyen la *estructura interna* de los objetos de una clase. En C++ un atributo también se denomina *dato miembro*. Para declarar un atributo, proceda exactamente igual que ha hecho para declarar cualquier otra variable en cualquier otra parte de un programa. Por ejemplo:

```
class Circulo
{
  private:
    double x, y;
    double radio;

  // ...
};
```

En una clase, cada atributo debe tener un nombre único. En cambio, se puede utilizar el mismo nombre con atributos, y con miembros en general, que pertenezcan a diferentes clases, porque una clase define su propio ámbito.

Es posible asignar un valor inicial a un atributo de una clase utilizando cualquiera de las formas expuestas en el apartado *Tipos, constantes, variables y estructuras* del capítulo *C++ versus C*. Por ejemplo, en la clase *Círculo* podemos iniciar el radio con el valor 1, aunque generalmente esto no es necesario, ya que como expondremos un poco más adelante este tipo de operaciones son típicas del constructor de la clase:

```
class Circulo
{
  private:
    double x, y;
    double radio = 1; // iniciación permitida
                      // también: double radio{ 1 }; (no ())
  // ...
};
```

También podemos declarar como atributos de una clase objetos de otras clases existentes. El siguiente ejemplo define la clase *Punto* y después declara el atributo *centro* de *Círculo*, de la clase *Punto*.

```
class Punto
{
```

```
  private:
    double x, y;

  public:
    Punto() {}
    Punto(double cx, double cy) { x = cx; y = cy; }
};

class Circulo
{
  private:
    Punto centro; // coordenadas del centro
    double radio; // radio del círculo
  // ...
};
```

Observe ahora que la clase *Círculo* tiene un atributo *centro* de la clase *Punto*, lo que implica definir previamente la clase *Punto*.

Un objeto de una clase no puede ser atributo de ella misma, a no ser que se declare **static**, pero sí puede serlo un puntero al objeto. Por ejemplo:

```
class Circulo
{
  private:
    // ...
    Circulo anterior;   // error: objeto de la misma clase
    Circulo* panterior; // correcto

  // ...
};
```

Métodos de una clase

Los métodos generalmente forman lo que se denomina *interfaz* o medio de acceso a la estructura interna de los objetos; ellos definen las operaciones que se pueden realizar con sus atributos (con el objeto). En C++ un método de una clase también se denomina *función miembro* de la clase. Desde el punto de vista de la POO, el conjunto de todos estos métodos se corresponde con el conjunto de mensajes a los que los objetos de una clase pueden responder.

Para definir un método de una clase, proceda exactamente igual que ha hecho para definir cualquier otro método (o función) en las aplicaciones realizadas en los capítulos anteriores. Recuerde también que los métodos no se pueden anidar. Como ejemplo puede observar los métodos *Círculo* (constructor con parámetros de la clase) y *longCircunferencia* de la clase *Círculo*.

```
class Circulo
{
  // ...
```

```
  public:
    Circulo(double cx, double cy, double r) // constructor
    {
      x = cx; y = cy;
      if (r < 0)
      {
        msgEsNegativo();
        r = -r;
      }
      radio = r;
    }

    double longCircunferencia()
    {
      return 2 * 3.1415926 * radio;
    }
    // ...
};
```

Control de acceso a los miembros de la clase

El concepto de clase incluye la idea de ocultación de datos, que básicamente consiste en que no se puede acceder directamente a los atributos de un objeto, sino que hay que hacerlo a través de métodos de su clase. Por ejemplo:

```
int main()
{
  Circulo c;    // invoca al constructor sin argumentos
  c.radio = 10; // error: radio es un atributo privado
}
```

El ejemplo anterior indica que la clase debería proporcionar un método público (**public**) que permitiera el acceso al atributo privado *radio*. Por ejemplo:

```
class Circulo
{
  // ...
  public:
    // ...
    void asignarRadio(double r)
    {
      if (r < 0)
        msgEsNegativo();
      else
        radio = r;
   }
};

int main()
{
  Circulo c;          // invoca al constructor sin argumentos
  c.asignarRadio(10); // asignar al objeto c el radio 10
}
```

Esto quiere decir que, de forma general, el usuario de la clase sólo tendrá acceso a uno o más métodos que le permitirán acceder a los miembros privados, ignorando la disposición de éstos (dichos métodos se denominan *métodos de acceso*). De esta forma se consiguen dos objetivos importantes:

1. Que el usuario no tenga acceso directo a la estructura de datos interna de la clase, para que no pueda generar código basado en esa estructura.

2. Que, si en un momento determinado alteramos la definición de la clase, excepto el prototipo de los métodos, todo el código escrito por el usuario basado en estos métodos no tendrá que ser retocado.

Piense que, si el objetivo uno no se cumpliera, cuando se diera el objetivo dos el usuario tendría que reescribir el código que hubiera desarrollado basándose en la estructura interna de los datos.

Por otra parte, si el usuario no tiene acceso directo a la estructura de datos interna del objeto, tampoco podrá asignar valores no permitidos, porque los métodos de acceso habrán sido diseñados para ello. Como ejemplo hemos visto que no se puede asignar un valor negativo al radio de un objeto *Circulo*.

Para controlar el acceso a los miembros de una clase, C++ provee las palabras clave **private** (privado), **protected** (protegido) y **public** (público), aunque también es posible omitirlas, en cuyo caso el acceso se supone privado. Estas palabras clave, denominadas *modificadores de acceso*, son utilizadas para indicar el tipo de acceso permitido a cada miembro de la clase. Si observamos la clase *Círculo* expuesta anteriormente, identificamos miembros privados, protegidos y públicos.

```
class Circulo
{
  // miembros privados
  private:
    double x, y;      // coordenadas del centro
    double radio;     // radio del círculo

  // miembros protegidos
  protected:
    void msgEsNegativo() { ... }

  // miembros públicos
  public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r) { ... } // constructor
    double longCircunferencia() { ... }
    double areaCirculo() { ... }
    void asignarRadio(double r) { ... }
};
```

Es importante no olvidar que un miembro de una clase sólo puede ser accedido, implícita o explícitamente, por un objeto de esa clase, utilizando el operador . (punto) o el operador -> cuando el objeto esté referenciado por un puntero. En el ejemplo siguiente, el método *areaCirculo* de la clase *Círculo* es accedido por el objeto *c* de la misma clase; en POO se dice que el objeto *c* recibe el mensaje *areaCirculo* y responde ejecutando el método del mismo nombre.

```
int main()
{
  Circulo c(100, 200, 10); // invoca al constructor y construye c
  double area = c.areaCirculo(); // c recibe el mensaje areaCirculo
  double lcir = longCircunferencia(); // error: función externa no
}                                     // declarada
```

Puede haber varias secciones privadas, protegidas o públicas. Cada una de ellas finaliza donde comienza la siguiente.

Acceso público

Un miembro de una clase declarado **public** (público) puede ser accedido por un objeto de esa clase en cualquier parte de la aplicación donde el objeto en cuestión sea accesible. Los miembros públicos de una clase constituyen la interfaz pública de los objetos de esa clase.

```
int main()
{
  Circulo c(100, 200, 10);
  double area = c.areaCirculo(); // correcto, miembro público
}
```

Una estructura (**struct**) es una clase cuyos miembros son públicos por omisión de los modificadores de acceso.

Acceso privado

Un miembro de una clase declarado **private** (privado) puede ser accedido por un objeto de esa clase sólo desde los métodos de dicha clase (vea más adelante *El puntero implícito* ***this***). Esto significa que no puede ser accedido por los métodos de cualquier otra clase, incluidas las subclases, ni por las funciones externas de la aplicación, como, por ejemplo, la función **main**.

```
int main()
{
  Circulo c(100, 200, 10);
  double r = c.radio; // error: miembro privado
}
```

Acceso protegido

Un miembro de una clase declarado **protected** (protegido) se comporta exactamente igual que uno privado para las funciones externas o para los métodos de cualquier otra clase, pero actúa como un miembro público para los métodos de sus subclases (véase el capítulo *Clases derivadas*).

Clases en archivos de cabecera

En el apartado *Compilación separada* del capítulo *C++ versus C* se estudió como, utilizando la técnica de compilación separada, se puede reutilizar, en un programa, código que ya está escrito. El siguiente ejemplo reutiliza la clase **string** que ya está escrita (se proporciona en la biblioteca de C++).

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

int main()
{
  string s = "abc";
  int n = s.size();
  // ...
}
```

Evidentemente, cuando el compilador C++ compile el código anterior necesitará saber qué es **string** para poder definir el objeto *s*, o cómo es el prototipo del método **size**, pero no necesita saber cómo está escrito el cuerpo de **size**; esa información es la que proporciona el archivo de cabecera, ya que el código implicado en el programa que se está compilando (como el código del método **size** de la clase **string**) será obtenido desde la biblioteca en la fase de enlace.

Resumiendo, cuando escribimos un programa, generalmente, reutilizamos código ya escrito que obtenemos de distintas bibliotecas escritas por otros; esto es, nosotros (los que escribimos los programas) somos los usuarios de esas bibliotecas, y, ¿quién escribió esas bibliotecas? La respuesta nos lleva a pensar que en este escenario intervienen, al menos, dos actores: 1) el que creó las bibliotecas y 2) el usuario que escribe programas utilizando esas bibliotecas. Estos dos actores serán suplantados por usted durante el estudio de esta obra: aprenderá a escribir clases para otros, pero tendrá que escribir programas que prueben esas clases.

Para escribir esas clases vamos a hacer uso de la técnica de *compilación separada*, esto es, organizaremos el código de cada una de nuestras clases en dos archivos: un archivo de cabecera (*.h*) que incluya la declaración de la clase (se trata de código fuente para incluir, **#include**, en otras unidades de traducción) y otro archivo (*.cpp*) que incluya las definiciones de los métodos de la clase; este segun-

do archivo, lo normal es que se proporcione compilado, por razones obvias. No obstante, mientras construimos nuestras clases, para probarlas como usuarios, por comodidad, incluiremos en el proyecto, además de los *.h*, los *.cpp* correspondientes, en lugar de los *.obj*.

Por ejemplo, volviendo a la clase *Círculo*, podríamos escribir la declaración de la misma en un archivo *circulo.h* así:

```
// circulo.h - Declaración de la clase Círculo
//
class Circulo
{
  // miembros privados
  private:
    double x, y;      // coordenadas del centro
    double radio;     // radio del círculo
  // miembros protegidos
  protected:
    void msgEsNegativo();
  // miembros públicos
  public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r); // constructor
    double longCircunferencia();
    double areaCirculo();
    void asignarRadio(double r);
};
```

Y las definiciones de los métodos en otro archivo fuente con extensión *.cpp*, por tratarse de una unidad de traducción. Por ejemplo, continuando con la clase *Círculo*, podríamos escribir la definición de sus métodos en un archivo *circulo.cpp* así:

```
// circulo.cpp - Definición de los métodos de la clase Círculo
#include <iostream>
#include "circulo.h"
using namespace std;

void Circulo::msgEsNegativo()
{
  cout << "El radio es negativo. Se convierte a positivo\n";
}

Circulo::Circulo(double cx, double cy, double r) // constructor
{
  x = cx; y = cy;
  if (r < 0)
  {
    msgEsNegativo();
    r = -r;
  }
  radio = r;
}
```

```
double Circulo::longCircunferencia()
{
  return 2 * 3.1415926 * radio;
}

double Circulo::areaCirculo()
{
  return 3.1415926 * radio * radio;
}

void Circulo::asignarRadio(double r)
{
  if (r < 0)
    msgEsNegativo();
  else
    radio = r;
}
```

Para que este archivo pueda ser compilado satisfactoriamente debe incluir el archivo de cabecera *circulo.h* (además de los necesarios de la biblioteca) que es donde está declarado el tipo *Circulo* al que el código a compilar hace referencia; esto es, el compilador para realizar la traducción necesita conocer los tipos y demás declaraciones a las que hace referencia el archivo fuente *.cpp*.

También, observamos que para definir un método fuera del cuerpo de la clase, hay que indicar a qué clase pertenece dicho método; de lo contrario, el compilador interpretará que se trata de una función externa (como es **main**), en vez de un método de una clase. Para ello hay que especificar el nombre de la clase antes del nombre del método, separado del mismo por el operador de ámbito (**::**). Esto es, en un programa que utilice la clase *Circulo* pueden coexistir las dos funciones siguientes:

```
void Circulo::asignarRadio(double r)
{
  // ...
}

void asignarRadio(double r)
{
  // ...
}
```

Mientras que la primera es una función miembro de la clase *Circulo* que, para un objeto concreto, puede acceder a todos los miembros de su clase, la segunda es una función externa (no pertenece a ninguna clase) que, para un objeto definido, por ejemplo, en el cuerpo de la misma, sólo podría acceder a la interfaz pública de ese objeto (esto es así, porque C++ es un lenguaje de programación híbrido, no es puro orientado a objetos).

Por otra parte, el hecho de especificar la clase a la que pertenece un método, define al método dentro del ámbito de esa clase, lo que permite que existan métodos con el mismo nombre en diferentes clases. Por ejemplo:

```
void Punto::asignarCoordenadaX(double d)
{
  // ...
}

void Circulo::asignarCoordenadaX(double d)
{
  // ...
}
```

Un archivo de cabecera, además de la declaración de la clase, debe contener también los métodos **inline** de la misma, para que el compilador pueda acceder al código fuente de los mismos y reemplazar con él, si procede, cada una de las llamadas a los mismos.

Los métodos definidos en el cuerpo de la clase son por definición **inline**.

También debemos pensar que un archivo fuente generalmente va contener varias directrices **#include**, lo que puede dar lugar a incluir en una unidad de traducción más de una copia de la definición de una clase y/o de otras declaraciones; por ejemplo, simplemente porque un archivo de cabecera incluido explícitamente por el desarrollador sea a su vez incluido por otro archivo de cabecera, hecho que generalmente pasará desapercibido. Esto provocaría errores de redefinición durante la compilación de ese módulo. Para evitar este problema, cada archivo de cabecera debe contener las directrices siguientes:

```
#if !defined ( _NOMBRE_H_ )
#define _NOMBRE_H_

// contenido del archivo de cabecera (nombre.h)
#endif // _NOMBRE_H_
```

donde `NOMBRE`, generalmente, coincide con el nombre del archivo de cabecera. El código anterior es equivalente a esto otro:

```
#pragma once
// contenido del archivo de cabecera (nombre.h)
```

Si aplicamos la teoría expuesta a la clase *Círculo*, vista anteriormente, obtendremos el siguiente resultado:

```
// circulo.h - Declaración de la clase Círculo
#if !defined( _CIRCULO_H_ )
#define _CIRCULO_H_
```

```
class Circulo
{
  // cuerpo de la clase
};
#endif // _CIRCULO_H_
```

Cuando un archivo de cabecera como el del ejemplo anterior es incluido en una unidad de traducción por primera vez, el símbolo `_NOMBRE_H_` no está definido; el preprocesador se dará cuenta de esto al evaluar la condición de la directriz **#if**; entonces, al ejecutar la directriz **#define**, lo define y, a continuación, incluye el archivo *nombre.h*. Si posteriormente tratamos de incluir el mismo archivo de cabecera, el símbolo `_NOMBRE_H_` ya está definido, lo que da lugar a que la condición de la directriz **#if** sea falsa y se ignore el contenido hasta **#endif**.

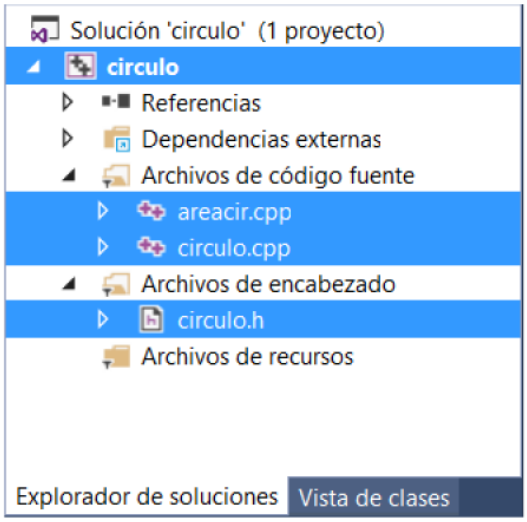
Posteriormente, para utilizar la clase *Círculo* simplemente tiene que incluir el archivo de cabecera *circulo.h* en los archivos fuente donde se haga referencia a ella. Lógicamente, cuando compile su aplicación, tiene que enlazar con la misma el archivo *circulo.obj* (resultado de compilar el archivo *circulo.cpp*) que contiene las definiciones de los métodos de la clase declarada en *circulo.h*. Tiene tres formas de enlazar este archivo: especificándolo directamente en la línea de órdenes; creando un archivo de proyecto (*.mak*) que incluya *circulo.obj* o *circulo.cpp* junto con los demás archivos de la aplicación o incluyéndolo en una biblioteca que sería referenciada en el proceso de compilación y enlace. Esta forma de trabajar permite que un usuario necesite conocer solamente la interfaz de la clase, sin importarle su implementación. Si posteriormente se modifica la clase *Círculo*, sin cambiar los prototipos de los métodos de la interfaz pública, lo único que tiene que hacer el usuario es volver a compilar su aplicación.

Como ejemplo, supongamos que escribimos el siguiente programa, que almacenamos en el archivo *areacir.cpp*:

```
// areacir.cpp - programa que utiliza la clase Círculo
#include <iostream>
#include "circulo.h"
using namespace std;

int main()
{
  Circulo c(100, 200, 10); // invoca al constructor y construye c
  double area = c.areaCirculo(); // c recibe el mensaje areaCirculo
  cout << area << endl;
}
```

Utilizando su EDI preferido, puede crear un proyecto, *circulo* por ejemplo, que incluya los archivos *circulo.h*, *circulo.cpp* (o *circulo.obj*) y *areacir.cpp*, y probar lo explicado hasta ahora:



IMPLEMENTACIÓN DE UNA CLASE

Según lo estudiado hasta ahora, la programación orientada a objetos con C++ sugiere escribir la declaración de cada clase en un archivo de cabecera y su definición en un archivo *.cpp*, fundamentalmente para reutilizar y mantener dicha clase posteriormente con facilidad. Como ejemplo, diseñaremos una clase que almacene una fecha, verificando que es correcta; esto es, que el día esté entre los límites 1 y días del mes, que el mes esté entre los límites 1 y 12 y que el año sea mayor o igual que 1582 (año gregoriano).

¿Por qué es útil definir una clase tan simple como esta? Porque si esta clase no existiera, cada usuario tendría que manipular las fechas directamente, o proporcionar funciones separadas para hacerlo, con lo que la noción de fecha estaría dispersa por todo el programa, lo cual sería más difícil de mantener, documentar o cambiar.

Parece lógico que la estructura de datos de un objeto fecha esté formada por los atributos *día*, *mes* y *año* y que permanezca oculta al usuario. Por otra parte, las operaciones sobre estos objetos tendrán que permitir, al menos, asignar una fecha, método *asignarFecha*, y obtener una fecha de un objeto existente, método *obtenerFecha*. Estos dos métodos formarán la interfaz pública. También, para verificar si la fecha que se quiere asignar es correcta, añadiremos un método protegido *fechaValida*. Cuando el día corresponda al mes de febrero, el método *fechaValida* necesitará comprobar si el año es bisiesto, para lo que añadiremos el método protegido *anyoBisiesto*. Al declarar protegidos estos dos métodos solo podrán ser accedidos desde la propia clase y desde una subclase de esta. Evidentemente se pueden añadir otros métodos, pero vamos a empezar con algo simple.

Según lo expuesto, podemos escribir una clase denominada *CFecha* así:

```
// fecha.h - Declaración de la clase CFecha
class CFecha
{
  // Atributos
  private:
    int dia, mes, anyo;
  // Métodos
  protected:
    bool anyoBisiesto(int aaaa);
    bool fechaValida(int dd, int mm, int aaaa);
  public:
    bool asignarFecha(int dd, int mm, int aaaa);
    void obtenerFecha(int& dd, int& mm, int& aaaa);
};
```

El método *anyoBisiesto* devuelve **true** si el año pasado como argumento es bisiesto; en otro caso, devuelve **false**.

El método *fechaValida* devuelve **true** si la fecha pasada como argumento es válida; en otro caso, devuelve **false**.

El método *asignarFecha* asigna, al objeto para el que es llamado/invocado, la fecha pasada como argumento. Devuelve **true** si la fecha pasada como argumento es válida; en otro caso, asigna una fecha predeterminada y devuelve **false**.

El método *obtenerFecha* recibe tres argumentos pasados por referencia para poder devolver en los mismos la fecha que representa el objeto para el que es llamado. No devuelve nada.

El paso siguiente es definir cada uno de esos métodos. Lo haremos en el archivo *fecha.cpp*. Al hablar de los modificadores de acceso quedó claro que cada uno de los métodos de una clase tiene acceso directo al resto de los miembros. Según lo expuesto, la definición del método *asignarFecha* puede escribirse así:

```
bool CFecha::asignarFecha(int dd, int mm, int aaaa)
{
  if (!fechaValida(dd, mm, aaaa))
  {
    cout << "Fecha incorrecta. Se asigna 01/01/2001.\n";
    dia = 1; mes = 1; anyo = 2001;
    return false;
  }
  dia = dd; mes = mm; anyo = aaaa;
  return true;
}
```

Observe que, por ser *asignarFecha* un método de la clase *CFecha*, puede acceder directamente a los atributos *día*, *mes* y *anyo* de su misma clase, independientemente de que sean privados. Estos atributos corresponderán en cada caso al objeto que recibe el mensaje *asignarFecha* (objeto para el que se invoca el méto-

do; vea más adelante, en este mismo capítulo, el puntero implícito **this**). Por ejemplo, si declaramos los objetos *fecha1* y *fecha2* de la clase *CFecha* y enviamos a *fecha1* el mensaje *asignarFecha*,

```
fecha1.asignarFecha(dd, mm, aaaa);
```

como respuesta a este mensaje, se ejecuta el método *asignarFecha* que asigna los datos *dd*, *mm* y *aaaa* al objeto *fecha1*; esto es, a *fecha1.dia*, *fecha1.mes* y *fecha1.anyo*; y si a *fecha2* le enviamos también el mensaje *asignarFecha*,

```
fecha2.asignarFecha(dd, mm, aaaa);
```

como respuesta a este mensaje, se ejecuta el método *asignarFecha* que asigna los datos *dd*, *mm* y *aaaa* al objeto *fecha2*; esto es, a *fecha2.dia*, *fecha2.mes* y *fecha2.anyo*.

Siguiendo las reglas enunciadas, finalizaremos el diseño de la clase escribiendo el resto de los métodos. El resultado que se obtendrá será la clase *CFecha* que se observa a continuación:

```
// fecha.cpp - Definición de los métodos de la clase CFecha
#include <iostream>
#include "fecha.h"
using namespace std;

bool CFecha::anyoBisiesto(int aaaa)
{
  return ((aaaa % 4 == 0) && (aaaa % 100 != 0) || (aaaa % 400 == 0));
}

bool CFecha::asignarFecha(int dd, int mm, int aaaa)
{
  if (!fechaValida(dd, mm, aaaa))
  {
    cout << "Fecha incorrecta. Se asigna 01/01/2001.\n";
    dia = 1; mes = 1; anyo = 2001;
    return false;
  }
  dia = dd; mes = mm; anyo = aaaa;
  return true;
}

void CFecha::obtenerFecha(int& dd, int& mm, int& aaaa)
{
  dd = dia; mm = mes; aaaa = anyo;
}

bool CFecha::fechaValida(int dd, int mm, int aaaa)
{
  bool diaCorrecto, mesCorrecto, anyoCorrecto;
  anyoCorrecto = (aaaa >= 1582); // ¿año correcto?
  mesCorrecto = (mm >= 1) && (mm <= 12); // ¿mes correcto?
```

```
  switch (mm)
  // ¿día correcto?
  {
    case 2:
      if (anyoBisiesto(aaaa))
        diaCorrecto = (dd >= 1 && dd <= 29);
      else
        diaCorrecto = (dd >= 1 && dd <= 28);
      break;
    case 4: case 6: case 9: case 11:
      diaCorrecto = (dd >= 1 && dd <= 30);
      break;
    default:
      diaCorrecto = (dd >= 1 && dd <= 31);
  }
  return diaCorrecto && mesCorrecto && anyoCorrecto;
}
```

Resumiendo: la funcionalidad de esta clase está soportada por los atributos privados *día*, *mes* y *anyo* y por los métodos *asignarFecha*, *obtenerFecha*, *fechaValida* y *anyoBisiesto*. Los dos primeros forman la interfaz pública de la clase.

MÉTODOS SOBRECARGADOS

En los capítulos anteriores, al trabajar con la biblioteca de C++ nos hemos encontrado con clases que implementan varias veces el mismo método. Por ejemplo, al hablar de la E/S dijimos que la clase **basic_istream** sobrecarga el operador >> con el fin de ofrecer al programador una notación cómoda que le permita obtener de la entrada estándar datos de cualquier tipo primitivo o derivado predefinido. Por ejemplo, algunas sobrecargas de este operador son:

```
basic_istream<...>& operator>>(int& n);
basic_istream<...>& operator>>(double& n);
basic_istream<...>& operator>>(basic_istream<...>& is, char* s);
```

También dijimos que la clase **basic_ostream** sobrecarga el operador << con el fin de ofrecer al programador una notación cómoda que le permita enviar a la salida estándar datos de cualquier tipo primitivo o derivado predefinido. Por ejemplo, algunas sobrecargas de este operador son:

```
basic_ostream<...>& operator<<(int n);
basic_ostream<...>& operator<<(double n);
basic_ostream<...>& operator<<(basic_ostream<...>& os, char* s);
```

¿En qué se diferencian estos métodos sobrecargados? En su número de parámetros y/o en el tipo de los mismos.

Pues bien, cuando en una clase un mismo método se define varias veces con distinto número de parámetros, o bien con el mismo número de parámetros pero

diferenciándose una definición de otra en que al menos un parámetro es de un tipo diferente, se dice que el método está *sobrecargado*.

Los métodos sobrecargados pueden diferir también en el tipo del valor retornado. Ahora bien, el compilador C++ no admite que se declaren dos métodos que sólo difieran en el tipo del valor retornado; deben diferir también en la lista de parámetros; esto es, lo que importa son el número y el tipo de los parámetros.

La sobrecarga de métodos elimina la necesidad de nombrar de forma diferente métodos que en esencia hacen lo mismo, o también hace posible que un método se comporte de una u otra forma según el número de argumentos con el que sea invocado, como es el caso de los métodos **operator<<** y **operator>>**.

Como ejemplo, sobrecargaremos el método *asignarFecha* para que pueda ser invocado con cero argumentos; con un argumento, el día; con dos argumentos, el día y el mes; y con tres argumentos, el día, el mes y el año. Los datos día, mes o año omitidos en cualquiera de los casos, serán obtenidos de la fecha actual proporcionada por el sistema. Añada los prototipos a *fecha.h*.

A continuación, escribimos en *fecha.cpp* estas sobrecargas sabiendo que la fecha actual del sistema se puede obtener a partir de las funciones **localtime** y **time** de la biblioteca de C. La función **localtime** utiliza una variable de tipo **static struct tm** para realizar la conversión, y lo que devuelve es la dirección de esa variable (ver el apéndice *B*).

```
bool CFecha::asignarFecha()
{
  // Por omisión, asignar la fecha actual.
  struct tm* fh;
  time_t segundos;

  time(&segundos);
  fh = localtime(&segundos);

  dia = fh->tm_mday;          // día de 1 a 31
  mes = fh->tm_mon + 1;       // tm_mon: mes de 0 a 11; enero = 0
  anyo = fh->tm_year + 1900; // tm_year: año - 1900
  return true;
}

bool CFecha::asignarFecha(int dd)
{
  asignarFecha();
  if (!fechaValida(dd, mes, anyo))
  {
    cout << "Fecha incorrecta. Se asigna 01/01/2001.\n";
    dia = 1; mes = 1; anyo = 2001;
    return false;
  }
  dia = dd;
```

```
  return true;
}

bool CFecha::asignarFecha(int dd, int mm)
{
  asignarFecha();
  if (!fechaValida(dd, mm, anyo))
  {
    cout << "Fecha incorrecta. Se asigna 01/01/2001.\n";
    dia = 1; mes = 1; anyo = 2001;
    return false;
  }
  dia = dd; mes = mm;
  return true;
}

bool CFecha::asignarFecha(int dd, int mm, int aaaa)
{
  if (!fechaValida(dd, mm, aaaa))
  {
    cout << "Fecha incorrecta. Se asigna 01/01/2001.\n";
    dia = 1; mes = 1; anyo = 2001;
    return false;
  }
  dia = dd; mes = mm; anyo = aaaa;
  return true;
}
```

Como se puede observar, el que una definición del método invoque a otra es una técnica de método abreviado que da como resultado métodos más cortos.

Por cada llamada al método *asignarFecha* que escribamos en un programa, el compilador C++ debe resolver cuál de los métodos con el nombre *asignarFecha* es invocado. Esto lo hace comparando el número y tipos de los argumentos especificados en la llamada con los parámetros especificados en las distintas definiciones del método. El siguiente ejemplo muestra las posibles formas de invocar al método *asignarFecha*:

```
fecha.asignarFecha();
fecha.asignarFecha(dd);
fecha.asignarFecha(dd, mm);
fecha.asignarFecha(dd, mm, aaaa);
```

Si el compilador C++ no encontrara un método exactamente con los mismos tipos de argumentos especificados en la llamada, realizaría sobre dichos argumentos las conversiones implícitas permitidas entre tipos, tratando de adaptarlos a alguna de las definiciones existentes del método. Si este intento fracasa, entonces se producirá un error.

ARGUMENTOS POR OMISIÓN

En ocasiones, nos podremos ahorrar escribir múltiples formas de un mismo método, si utilizamos parámetros con valores por omisión (se entiende, por omisión de los argumentos en la llamada al método). Esto ocurrirá cuando partiendo de un método con una lista de *n* parámetros preestablecidos, tengamos la necesidad de pasar 0 a *n* argumentos. El ejemplo anterior se ajusta perfectamente a lo expuesto. En este caso, podemos sustituir todas las formas anteriores de *asignarFecha* por una única forma del método cuyo prototipo sea, por ejemplo, así:

```
bool asignarFecha(int dd = 0, int mm = 0, int aaaa = 0);
```

La definición de este método puede ser como se indica a continuación:

```
bool CFecha::asignarFecha(int dd, int mm, int aaaa)
{
  struct tm* fh;
  time_t segundos;

  time(&segundos);
  fh = localtime(&segundos);

  if (aaaa == 0 && mm == 0 && dd == 0) // cero argumentos
    dd = fh->tm_mday; // día de 1 a 31
  if (aaaa == 0 && mm == 0) // un argumento
    mm = fh->tm_mon + 1; // mes de 0 a 11; enero = 0
  if (aaaa == 0) // dos argumentos
    aaaa = fh->tm_year + 1900; // año - 1900

  if (!fechaValida(dd, mm, aaaa))
  {
    cout << "Fecha incorrecta. Se asigna 01/01/2001.\n";
    dia = 1; mes = 1; anyo = 2001;
    return false;
  }
  dia = dd; mes = mm; anyo = aaaa;
  return true;
}
```

PROBAR LA CLASE

Para comprobar que la clase *CFecha* que acabamos de diseñar trabaja correctamente, podemos crear un proyecto que incluya los archivos *fecha.h* y *fecha.cpp* que acabamos de escribir y un archivo *test.cpp* que incluya, además de la función **main**, todas las funciones externas que creamos necesarias para construir una aplicación que permita verificar todos los métodos de la clase *CFecha*. Según esto, solo nos queda escribir este archivo *test.cpp*, que puede ser así:

```
// test.cpp - Trabajar con la clase CFecha
#include <iostream>
#include "fecha.h"
using namespace std;
void leerFecha(int&, int&, int&);
void visualizarFecha(CFecha& fecha);

int main()
{
  CFecha fecha; // objeto de tipo CFecha
  int dd = 0, mm = 0, aaaa = 0;
  bool fechaValida = true;
  do
  {
    leerFecha(dd, mm, aaaa);
    fechaValida = fecha.asignarFecha(dd, mm, aaaa);
  }
  while (!fechaValida);
  visualizarFecha(fecha);
}

void leerFecha(int& dia, int& mes, int& anyo)
{
  cout << "día:  "; cin >> dia;
  cout << "mes:  "; cin >> mes;
  cout << "anyo: "; cin >> anyo;
}

void visualizarFecha(CFecha& fecha)
{
  int dd, mm, aaaa;
  fecha.obtenerFecha(dd, mm, aaaa);
  cout << dd << "/" << mm << "/" << aaaa << "\n";
}
```

Notar que la clase *CFecha* declara los atributos *día*, *mes* y *anyo* privados. Esto quiere decir que sólo son accesibles por los métodos de su clase. Si un método de otra clase, o bien una función externa, intenta acceder a uno de estos atributos, el compilador genera un error. Por ejemplo:

```
int main()
{
  CFecha fecha; // objeto de tipo CFecha
  int dd = 0, mm = 0, aaaa = 0;
  // ...
  int dd = fecha.dia:  // error: día es un miembro privado
  fecha.mes = 1;       // error: mes es un miembro privado
}
```

En cambio, los métodos *asignarFecha* y *obtenerFecha* son públicos. Por lo tanto, son accesibles, además de por los métodos de su clase, por cualquier otro método de otra clase o función externa. Sirva como ejemplo la función *visualizarFecha* de esta aplicación. Esta función presenta en la salida estándar la fecha almacenada en el objeto que se le pasa como argumento. Observe que tiene que