INTRODUCCIÓN

La modularización de un programa utiliza la noción de *tipo abstracto de dato* (**TAD**) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado **TAD**.

Una **clase** es un tipo de dato que contiene código (métodos) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, tal como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En el capítulo se aprenderá a crear (definir y especificar) y utilizar clases individuales.

2.1. CLASES Y OBJETOS

Las tecnologías orientadas a objetos combinan la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas, son los elementos clave sobre los que se articula la orientación a objetos.

2.1.1. ¿Qué son objetos?

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, tales como posición, tamaño, color, forma, textura, etc., que definen su estado. Los objetos también tienen ciertos comportamientos que los hacen diferentes de otros objetos.

Booch[1], define un *objeto* como "algo que tiene un estado, un comportamiento y una identidad". Supongamos una máquina de una fábrica. El *estado* de la *máquina* puede estar *funcionando/parada* ("on/of"), su potencia, velocidad máxima, velocidad actual, temperatura, etc. Su *comportamiento* puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, condiciones de señal de error o cambiar la velocidad. Su *identidad* se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar en el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En un diseño de un programa orientado a objetos, se crea una abstracción (un modelo simplificado) de la máquina basado en las propiedades y comportamiento que son útiles en el tiempo.

Martin/Odell definen un objeto como "cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos".

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una

---

[1] Booch, Grady. *Análisis y diseño orientado a objetos con aplicaciones*. Madrid : Díaz de Santos/Addison-Wesley, 1995. (libro traducido del inglés por Luis Joyanes, coautor de esta obra).

interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico tal como una máquina de fax tiene una interfaz de usuario bien definida; esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón "enviar". El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles.

2.1.2. ¿Qué son clases?

En términos prácticos, una **clase** es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch denomina a una clase como "un conjunto de objetos que comparten una estructura y comportamiento comunes".

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios* o *métodos*. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos, variables* o *variables instancia*. El término *atributo* se utiliza en análisis y diseño orientado a objetos y el término *variable instancia* se suele utilizar en programas orientados a objetos.

2.2. DECLARACIÓN DE UNA CLASE

Antes de que un programa pueda crear objetos de cualquier clase, la clase debe ser *declarada* y los métodos definidos (implementados). La declaración de una clase significa dar a la misma un nombre, darle nombre a los elementos que almacenan sus datos y describir los métodos que realizarán las acciones consideradas en los objetos.

Formato


```
class NombreClase
{
     lista_de_miembros
};
```

`NombreClase`	Nombre definido por el usuario que identifica a la clase (puede incluir letras, números y subrayados como cualquier identificador).
`lista_de_miembros`	Datos y funciones miembros de la clase.

Las *declaraciones* o *especificaciones* no son código de programa ejecutable. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer los métodos que utilizará el programa. En C++ los métodos se denominan funciones miembro, normalmente en la declaración sólo se escribe el prototipo de la función. Las declaraciones de las clases se sitúan en archivos `.h` (`NombreClase.h`) y la implementación de las funciones miembro en el archivo `.cpp` (`NombreClase.cpp`).

EJEMPLO 2.1. Definición de una clase llamada `Punto` que contiene las coordenadas *x* e *y* de un punto en un plano.

La declaración de la clase se guarda en el archivo *Punto.h*:

```
//archivo Punto.h
class Punto
{
private:
   int x, y;                        // coordenadas x, y
public:
   Punto(int x_, int y_)            // constructor
   {
       x = x_;
       y = y_;
   }
   Punto() { x = y = 0;}            // constructor sin argumentos
   int leerX() const;               // devuelve el valor de x

   int leerY() const;               // devuelve el valor de y
   void fijarX(int valorX)          // establece el valor de x
   void fijarY(int valorY)          // establece el valor de y
};
```

La definición de las funciones miembro se realiza en el archivo *Punto.cpp*:

```
#include "Punto.h"
int Punto::leerX() const
{
    return x;
}
int Punto::leerY() const
{
    return y;
}
void Punto::fijarX(int valorX)
{
    x = valorX;
}
void Punto::fijarY(int valorY)
{
    y = valorY;
}
```

2.2.1. Objetos

Una vez que una clase ha sido definida, un programa puede contener una *instancia* de la clase, denominada un *objeto de la clase*. Un objeto se crea de forma estática, de igual forma que se define una variable. También de forma dinámica, con el operador `new` aplicado a un constructor de la clase. Por ejemplo, un objeto de la clase `Punto` inicializado a las coordenadas `(2,1)`:

```
Punto p1(2, 1);                  // objeto creado de forma estática
Punto* p2 = new Punto(2, 1);     // objeto creado dinámicamente
```

Formato para crear un objeto

```
NombreClase varObj(argumentos_constructor);
```

Formato para crear un objeto dinámico

```
NombreClase* ptrObj;
ptrObj = new NombreClase(argumentos_constructor);
```

Toda clase tiene una o más funciones miembro, denominadas constructores, para inicializar el objeto cuando es creado; tienen el mismo nombre que el de la clase, no tienen tipo de retorno y pueden estar sobrecargados.

El *operador de acceso* a un miembro del objeto, selector punto (.), selecciona un miembro individual de un objeto de la clase. Por ejemplo:

```
Punto p2;                              // llama al constructor sin argumentos
p2.fijarX(10);
cout << " Coordenada  x es " << p2.leerX();
```

El otro *operador de acceso* es el selector *flecha* (->), selecciona un miembro de un objeto desde un puntero a la clase. Por ejemplo:

```
Punto* p;
p = new Punto(2, -5);                  // crea objeto dinámico
cout << " Coordenada  y es " << p -> leerY();
```

2.2.2. Visibilidad de los miembros de la clase

Un principio fundamental en programación orientada a objetos es la *ocultación de la información*. Significa que determinados datos del interior de una clase no se puede acceder por funciones externas a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o métodos privados sólo se puede acceder desde dentro de la clase. Por el contrario, los datos o métodos públicos son accesibles desde el exterior de la clase (véase la Figura 2.1).

Para controlar el acceso a los miembros de la clase se utilizan tres diferentes *especificadores de acceso:* `public`, `private` y `protected`. Cada miembro de la clase está precedido del especificador de acceso que le corresponde.

Formato

```
class NombreClase
{
private:
     declaraciones de miembros privados;
protected:
     declaraciones de miembros protegidos;
```

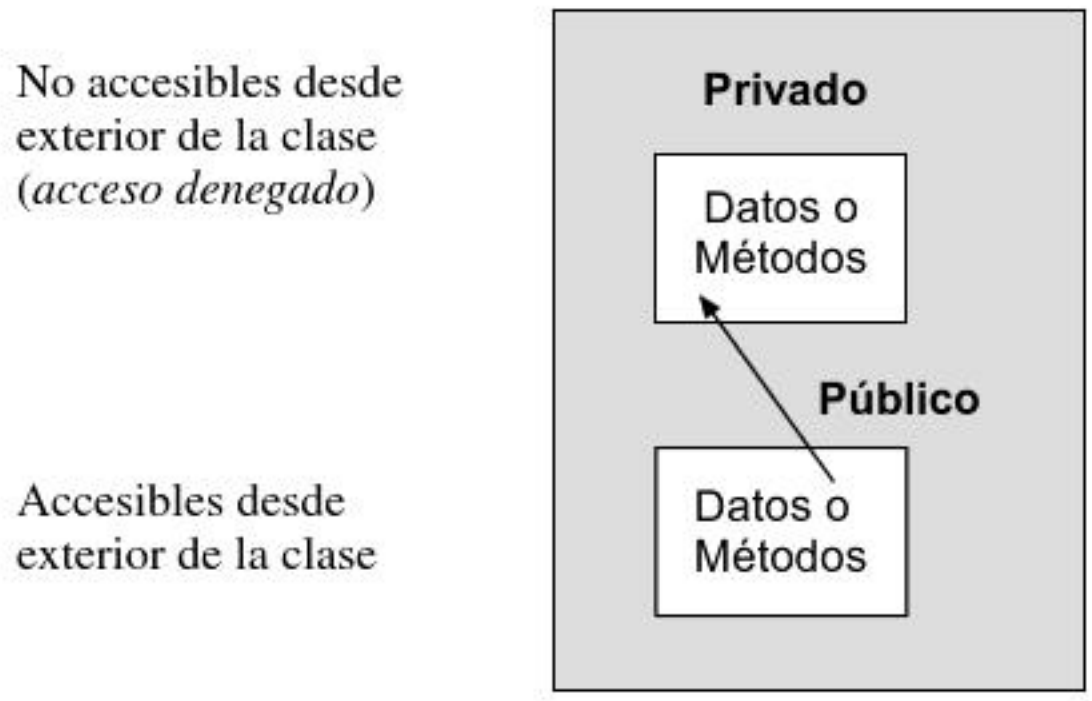

Figura 2.1. Secciones pública y privada de una clase.


```
public:
     declaraciones de miembros públicos;
};
```

Por omisión, los miembros que aparecen a continuación de la llave de inicio de la clase, `{`, son privados. A los miembros que siguen a la etiqueta `private` sólo se puede acceder por funciones miembro de la misma clase. A los miembros `protected` sólo se puede acceder por funciones miembro de la misma clase y de las clases derivadas. A los miembros que siguen a la etiqueta `public` se puede acceder desde dentro y desde el exterior de la clase. Las secciones `public`, `protected` y `private` pueden aparecer en cualquier orden.

EJEMPLO 2.2. Declaración de la clase `Foto` y `Marco` con miembros declarados con distinta visibilidad.

```
class Foto
{
private:
     int nt;
     char opd;
protected:
     string q;
public:
     Foto(string r)     // constructor
     {
          nt = 0;
          opd = 'S';
          q = r;
     }
     double mtd();
};

class Marco
{
private:
     double p;
     string t;
```

```
public:
     Marco();       // constructor
     void poner()
     {
          Foto* u = new Foto("Paloma");
          p = u -> mtd();
          t = "**" + u -> q + "**";
     }
};
```

Tabla 2.1. Visibilidad, "x" indica que el acceso está per mitido

Tipo de miembro	Miembro de la misma clase	Miembro de una clase derivada	Miembro de otra clase (externo)
`private`	x		
`protected`	x	x	
`public`	x	x	x

Aunque las secciones *públicas*, *privadas* y *protegidas* pueden aparecer en cualquier orden, los programadores suelen seguir ciertas reglas en el diseño que citamos a continuación, y que usted puede elegir la que considere más eficiente.

1. Poner los miembros privados primero, debido a que contiene los atributos (datos).
2. Se pone los miembros públicos primero debido a que los métodos y los constructores son la interfaz del usuario de la clase.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El *principio de ocultación* de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permite que los detalles de implementación de los objetos sean ignorados. Por consiguiente, los datos y métodos públicos forman la interfaz externa del objeto, mientras que los elementos *privados* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.

El principio de *encapsulamiento* significa que las estr ucturas de datos inter nas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

2.2.3. Funciones miembro de una clase

La declaración de una clase incluye la declaración o prototipo de las funciones miembros (métodos). Aunque la implementación se puede incluir dentro del cuerpo de la clase (`inline`), normalmente se realiza en otro archivo (con extensión `.cpp`) que constituye la definición de la clase. La Figura 2.2 muestra la declaración de la clase `Producto`.

```
class Producto
{
protected:
   int numProd;
   string nomProd;
   string descripProd;
private:
   double precioProd;
public:
   Producto();
   void verProducto();
   double obtenerPrecio();
   void actualizarProd(int b);
};
```

nombre de la clase
acceso para almacenamiento de datos
declaraciones para almacenamiento de datos
funciones miembro (métodos)

Figura 2.2. Definición típica de una clase.

EJEMPLO 2.3. La clase `Racional` representa un número racional. Por cada dato, numerador y denominador, se proporciona una función miembro que devuelve su valor y otra función para asignar numerador y denominador. Tiene un constructor que inicializa un objeto a 0/1.

En esta ocasión las funciones miembro se implementan directamente en el cuerpo de la clase.

```
// archivo Racional.h
class Racional
{
private:
    int numerador;
    int denominador;
public:
    Racional()
    {
        numerador = 0;
        denominador = 1;
    }
    int leerN() const { return numerador; }
    int leerD() const { return denominador; }
    void fijar (int n, int d)
    {
        numerador = n;
        denominador = d;
    }
};
```

2.2.4. Funciones en línea y fuera de línea

Las funciones miembro definidas dentro del cuerpo de la declaración de la clase se denominan definiciones de funciones *en línea* (`inline`). Para el caso de funciones más grandes, es preferible codificar sólo el prototipo de la función dentro del bloque de la clase y codificar la im-

plementación de la función en el exterior. Esta forma permite al creador de una clase ocultar la implementación de la función al usuario de la clase proporcionando sólo el código fuente del archivo de cabecera, junto con un archivo de implementación de la clase precompilada.

En el siguiente ejemplo, `FijarEdad()` de la clase `Lince` se declara pero no se define en la declaración de la clase:

```
class Lince
{
public:
   void FijarEdad(int a);
private:
   int edad;
   string habitat;
};
```

La implementación de una función miembro externamente a la declaración de la clase, se hace en una definición de la función *fuera de línea*. Su nombre debe ser precedido por el nombre de la clase y el signo de puntuación :: denominado *operador de resolución de ámbito*. El operador :: permite al compilador conocer que `FijarEdad()` pertenece a la clase `Lince` y es, por consiguiente, diferente de una función global que pueda tener el mismo nombre o de una función que tenga ese nombre que puede existir en otra clase. La siguiente función global, por ejemplo, puede coexistir dentro del mismo ámbito que `Lince::FijarEdad()`:

```
// función global:
void FijarEdad(int valx)
{
   // ...
}
// función en el ámbito de Lince:
void Lince::FijarEdad(int a)
{
    edad = a;
}
```


2.2.5. La palabra reservada `inline`

La decisión de elegir funciones en línea y fuera de línea es una cuestión de eficiencia en tiempo de ejecución. Una función en línea se ejecuta normalmente más rápida, ya que el compilador inserta una copia «fresca» de la función en un programa en cada punto en que se llama a la función. La definición de una función miembro en línea no garantiza que el compilador lo haga realmente en línea; es una decisión que el compilador toma, basado en los tipos de las sentencias dentro de la función y cada compilador de C++ toma esta decisión de modo diferente.

Si una función se compila en línea, se ahorra tiempo de la UCP (CPU) al no tener que ejecutar una instrucción "*call*" (llamar) para bifurcar a la función y no tener que ejecutar una instrucción `return` para retornar al programa llamador. Si una función es corta y se llama cientos de veces, se puede apreciar un incremento en eficiencia cuando actúa como función en línea.

Una función localizada fuera del bloque de la definición de una clase se puede beneficiar de las ventajas de las funciones en línea si está precedida por la palabra reservada `inline`:

```
inline void Lince::FijarEdad(int a)
{
    edad = a;
}
```

Dependiendo de la implementación de su compilador, las funciones que utilizan la palabra reservada `inline` se puede situar en el mismo archivo de cabecera que la definición de la clase. Las funciones que no utilizan `inline` se sitúan en el mismo módulo de programa, pero no el archivo de cabecera. Estas funciones se sitúan en un archivo `.cpp`.

Ejercicio 2.1

Definir una clase `DiaAnyo` con los atributos `mes` y `día`, los métodos `igual()` y `visualizar()`. El mes se registra como un valor entero (1, Enero, 2, Febrero, etc.). El día del mes se registra en otra variable entera día. Escribir un programa que compruebe si una fecha es su cumpleaños.

La función *principal*, `main()`, crea un objeto `DiaAnyo` y llama al método `igual()` para determinar si coincide la fecha del objeto con la fecha de su cumpleaños, que se ha leído de dispositivo de entrada.

```
// archivo DiaAnyo.h

class DiaAnyo
{
private:
   int dia, mes;
public:
    DiaAnyo(int d, int m);
    bool igual(const DiaAnyo& d) const;
    void visualizar() const;
};
```


La implementación de las funciones miembro se guarda en el archivo `DiaAnyo.cpp`:

```
#include <iostream>
using namespace std;
#include "DiaAnyo.h"

DiaAnyo::DiaAnyo(int d, int m)
{
    dia = d;
    mes = m;
}

bool DiaAnyo::igual(const DiaAnyo& d)const
{
    if ((dia == d.dia) && (mes == d.mes))
        return true;
    else
        return false;
}
```

```
void DiaAnyo::visualizar() const
{
    cout << "mes = " << mes << " , dia = " << dia << endl;
}
```

Por último, el archivo `DemoFecha.cpp` contiene la función `main()`, crea los objetos y se envian mensajes.

```
#include <iostream>
using namespace std;
#include "DiaAnyo.h"

int main()
{
    DiaAnyo* hoy;
    DiaAnyo* cumpleanyos;
    int d, m;

    cout << "Introduzca fecha de hoy, dia: ";
    cin >> d;
    cout << "Introduzca el número de mes: ";
    cin >> m;
    hoy = new DiaAnyo(d, m);
    cout << "Introduzca su fecha de nacimiento, dia: ";
    cin >> d;
    cout << "Introduzca el número de mes: ";
    cin >> m;
    cumpleanyos = new DiaAnyo(d, m);
    cout << " La fecha de hoy es ";
    hoy -> visualizar();
    cout << " Su fecha de nacimiento es ";
    cumpleanyos -> visualizar();
    if (hoy -> igual(*cumpleanyos))
        cout << "¡Feliz cumpleaños ! " << endl;
    else
        cout << "¡Feliz dia ! " << endl;
    return 0;
}
```


2.2.6. Sobrecarga de funciones miembro

Al igual que sucede con las funciones no miembro de una clase, las funciones miembro de una clase se pueden sobrecargar. Una función miembro se puede sobrecargar pero sólo en su propia clase.

Las mismas reglas utilizadas para sobrecargar funciones ordinarias se aplican a las funciones miembro: dos miembros sobrecargados no puede tener el mismo número y tipo de parámetros. La sobrecarga permite utilizar un mismo nombre para una función y ejecutar la función definida más adecuada a los parámetros pasados durante la ejecución del programa. La ventaja fundamental de trabajar con funciones miembro sobrecargadas es la comodidad que aporta a la programación.