

# Tema V

## Clases (Segunda parte)

### *Objetivos:*

- ▷ Ver cómo gestiona C++ el copiado entre objetos y sus implicaciones en el diseño de clases.
- ▷ Empezar a diseñar clases que colaboran entre sí para poder resolver problemas más complejos.

**Nota:** En este tema usaremos preferentemente clases. Todo lo que veamos también se aplica a los `struct` ya que, en C++, únicamente se diferencian de las clases en que el ámbito por defecto es `public` en vez de `private`.

## V.1. Copiando objetos

### V.1.1. Operador de asignación

#### V.1.1.1. Funcionamiento del operador de asignación

C++ permite asignar objetos entre sí a través del *operador de asignación por defecto (default assignment operator)* =

Este operador lo define automáticamente el compilador para cualquier clase, y funciona como si trabajásemos con tipos básicos:

```
objeto = otro_objeto;
```

- ▷ Ambos objetos han de ser de la misma clase.
- ▷ Se realiza una copia de **todos** los datos miembro, **tanto los privados como los públicos**.
- ▷ No tiene sentido hablar de *copiar* los métodos. ¡Ya estaban disponibles!

*El operador de asignación por defecto permite asignar objetos entre sí. Se copia el estado del objeto.*

```
class MiClase{
private:
    int privada;
public:
    int publica;

    MiClase(int priv, int pub)
        :privada(priv),
        publica(pub)
    {
    }
    int Privada(){
        return privada;
    }
    int Publica(){
        return publica;
    }
};

int main(){
    MiClase uno(6,7), otro(5,8);

    otro = uno;
    cout << "\nPrivada de uno: " << uno.Privada();    // 6
    cout << "\nPública de uno: " << uno.Publica();    // 7
    cout << "\nPrivada de otro: " << otro.Privada(); // 6
    cout << "\nPública de otro: " << otro.Publica(); // 7
}
```

**Nota.** Recordemos que jamás usaremos datos miembro públicos. El anterior es un ejemplo para mostrar los efectos del operador de asignación.

### V.1.1.2. El operador de asignación y los datos miembro constantes

¿Qué pasa cuando una clase contiene datos miembro constantes?

- Una constante estática es la misma para todos los objetos de una clase. Por tanto, éstas no dan problemas en la asignación entre objetos.

```
class SecuenciaCaracteres{
private:
    static const int DIM = 50;
    char vector_privado[DIM];
    int total_utilizados;
public:
    SecuenciaCaracteres()
        :total_utilizados(0)
    {
    }
    .....
};

int main(){
    SecuenciaCaracteres secuencia, otra_secuencia;

    secuencia.Aniade('t');
    secuencia.Aniade('e');
    secuencia.Aniade('c');
    secuencia.Aniade('a');
    otra_secuencia = secuencia;

    // Es posible ya que la constante
    // es estática, es decir, la misma
    // para todos los objetos
```

- ▷ Una constante no estática es propia de cada objeto. ¿Qué sentido tiene cambiar su valor con la constante de otro objeto? Ninguno.

***C++ NO permite asignar objetos entre sí (usando el operador de asignación por defecto) cuando la clase contiene datos miembro constantes no estáticos***

```
class CuentaBancaria{
private:
    const string identificador;
    .....
public:
    CuentaBancaria(string identificador_cuenta, double saldo_inicial)
        :saldo(saldo_inicial),
        identificador(identificador_cuenta)
    {
    }
    .....
};

int main(){
    CuentaBancaria cuenta("Un identificador", 20000);
    CuentaBancaria otra_cuenta("Otro identificador", 30000);

    otra_cuenta = cuenta; // Error de compilación, al tener
                          // la clase datos miembro constantes
```

## V.1.2. El constructor de copia

**C++ permite inicializar un objeto con los datos de otro:**

```
class MiClase{  
public:  
    MiClase (int parametro){ ..... }  
    .....  
};  
  
int main(){  
    MiClase ya_creado(9), otro_ya_creado(8);  
  
    MiClase nuevo(ya_creado);  
  
    otro_ya_creado = ya_creado;
```

**<- Constructor de copia**

**<- Operador de asignación**

En la sentencia `MiClase nuevo(ya_creado);` el compilador llama automáticamente al **constructor de copia (copy constructor)** . Es un constructor que recibe como parámetro un objeto de la misma clase y es proporcionado automáticamente (de oficio) por el compilador.

El constructor de copia realiza la asignación de los datos miembro que tenía `ya_creado` a `nuevo`.

*El constructor de copia permite inicializar, en el mismo momento de su definición, el estado de un objeto con los datos de otro objeto (se copia su estado).*

```
int main(){
    SecuenciaCaracteres secuencia;

    secuencia.Aniade('d');
    secuencia.Aniade('o');
    secuencia.Aniade('s');

    SecuenciaCaracteres otra_secuencia(secuencia);
    // Ambas secuencias contienen {'d','o','s'}
    .....
```

---

### En resumen:

- ▷ C++ proporciona **dos constructores de oficio**:
  - Constructor sin parámetros.
  - Constructor de copia.
- ▷ El programador puede definir:
  - Constructor con/sin parámetros, en cuyo caso ya no está disponible el constructor sin parámetros de oficio proporcionado por el compilador.
  - Constructor de copia, en cuyo caso ya no está disponible el constructor de copia de oficio proporcionado por el compilador (se verá en el segundo cuatrimestre)

***Si el programador define un constructor, ya no estará disponible el correspondiente constructor de oficio***

## Diferencias entre el operador de asignación y el constructor de copia:

- ▷ El operador de asignación es invocado cuando ya tenemos creados dos objetos y ejecutamos la sentencia `objeto_1 = objeto_2;`
- ▷ El constructor de copia es invocado en el momento de la declaración del objeto, justo en el momento en el que se va a crear éste.  
Veremos otros sitios en los que el compilador también invoca automáticamente al constructor de copia.

Si un objeto tiene datos constantes, **sí** podemos inicializar otro objeto con los datos del primero (porque todavía no tenía definida la constante):

```
class CuentaBancaria{
private:
    const string identificador;
    .....
};

int main(){
    CuentaBancaria cuenta("Constante a nivel de objeto", 20000);
    CuentaBancaria otra_cuenta(cuenta);           // Correcto
    otra_cuenta = cuenta;                          // Error de compilación
```



## V.1.3. Vectores de objetos

Para crear un vector de objetos, se usa la notación habitual:

```
const int TAMANIO = 50;
MiClase vector_de_objetos[TAMANIO];
```

Recordemos que el compilador crea un objeto cuando se declara:

```
class MiClase{
.....
};
int main(){
    MiClase objeto;
```

Lo mismo ocurre con las componentes de un vector; se crean en el momento de definir el vector:

```
class MiClase{
.....
};
int main(){
    const int TAMANIO = 50;
    MiClase vector_de_objetos[TAMANIO]; // <- Se crean 50 objetos
```

***Cuando se define un vector de objetos, se crean automáticamente tantos objetos como tamaño tenga el vector.***

**Si la clase no tiene constructor sin parámetros, hay que crear los objetos en la misma declaración del vector:**

```
class MiClase{
public:
    MiClase(int parametro){
        .....
    }
};

int main(){
    const int TAMANIO = 2;

    MiClase vector_de_objetos[TAMANIO]; // Error de compilación

    MiClase un_objeto(5);
    MiClase otro_objeto(6);
    MiClase vector_de_objetos[TAMANIO] = {un_objeto, otro_objeto};
    .....
}
```



***Para simplificar, en esta asignatura, supondremos que si trabajamos con un vector de objetos, la clase proporciona un constructor sin parámetros:***

- ▷ ***O bien porque no tenga ningún constructor definido, y sea el proporcionado de oficio por el compilador.***
- ▷ ***O bien porque esté definido explícitamente en la clase.***

## V.2. Métodos y objetos

### V.2.1. Objetos y funciones

En lo que resta de asignatura, no definiremos funciones globales que actúen sobre objetos. Este diseño puede aceptarse en algunos casos (se verá con el uso de *iteradores (iterators)* , por ejemplo, en el segundo cuatrimestre) pero en esta asignatura nunca lo usaremos.

Así pues, no definiremos funciones del tipo:

```
class Punto2D{
    .....
};
class Recta{
    .....
};

bool Contiene(Recta una_recta, Punto2D un_punto){
    .....
}
```



Lo que haremos será definir un método de la clase Recta:

```
class Punto2D{
    .....
};
class Recta{
    .....
    bool Contiene(Punto2D un_punto){
        .....
    }
};
```



**Evite por ahora la definición de funciones globales a las que se les pase objetos como parámetros. Debemos definir métodos que se ejecuten sobre los objetos.**

**Permitiremos el uso de funciones globales cuando éstas sean muy genéricas y sólo se les pase como parámetros tipos simples.**

**IMPORTANT**

**Consulte también el apartado de la página 640**

## V.2.2. Pasando objetos como parámetros a los métodos

*Los objetos, al igual que los datos de tipos simples, se pueden pasar como parámetros y ser devueltos por los métodos. Es más, los métodos de una clase podrán recibir como parámetros y devolver objetos de esa misma clase.*

```
class MiClase{
    .....
};
class OtraClase{
    void UnMetodo(MiClase parametro_formal){
        .....
    };
int main(){
    MiClase parametro_actual(6,7);
    OtraClase nuevo_objeto;
    nuevo_objeto.UnMetodo(parametro_actual);
}
```

El parámetro formal recibirá una *copia* del actual. Esta copia `parametro_formal(parametro_actual)` se realiza a través del constructor de copia ya que el parámetro formal se crea en el momento de la llamada al método.

*Al pasar los objetos como parámetros, se invoca automáticamente al constructor de copia*

**Ejemplo.** La ecuación de una recta viene dada en la forma

$$Ax + By + C = 0$$

es decir, todos los puntos  $(x, y)$  que satisfacen dicha ecuación forman una recta.

Queremos definir la clase `Recta` y ver si contiene a un punto.

**Cabecera:**

Punto2D	
- double	abscisa
- double	ordenada
+	Punto2D(double abscisaPunto, double ordenadaPunto)
+ void	SetCoordenadas (double abscisaPunto, double ordenadaPunto)
+ double	Abscisa()
+ double	Ordenada()

Recta	
- double	A
- double	B
- double	C
- bool	SonCorrectos(double coef_x, double coef_y)
+	Recta(double coef_x, double coef_y, double coef_indep)
+ void	SetCoeficientes(double coef_x, double coef_y, double coef_indep)
+ double	CoeficienteA()
+ double	CoeficienteB()
+ double	CoeficienteC()
+ double	Ordenada_en(double x)
+ double	Abscisa_en(double y)
+ double	Pendiente()
+ bool	Contiene(Punto2D punto)

### **Llamada:**

```
int main(){
    .....
    contiene = una_recta.Contiene(un_punto);

    if (contiene)
        cout << "\nLa recta contiene el punto";
    else
        cout << "\nLa recta no contiene el punto";
}
```

### **Implementación:**

```
bool SonIguales(double uno, double otro) {
    return abs(uno-otro) <= 1e-6;
}

class Punto2D{
    .....
};

class Recta{
    .....
    bool Contiene(Punto2D punto){
        double ordenada_recta = Ordenada_en(punto.Abscisa());

        return SonIguales(ordenada_recta , punto.Ordenada());

        // O también:
        // return SonIguales(A * punto.Abscisa() +
        //      B * punto.Ordenada() + C , 0.0);
    }
};
```

[http://decsai.ugr.es/jccubero/FP/V\\_punto\\_recta\\_segmento.cpp](http://decsai.ugr.es/jccubero/FP/V_punto_recta_segmento.cpp)

**Ejemplo.** Sobre la clase `SecuenciaCaracteres`, añadir un método para borrar un conjunto de posiciones dadas por una secuencia de enteros.

```
secuencia --> {'a','b','c','d','e','f'}  
posiciones_a_borrar --> {1, 3}  
secuencia --> {'a','c','e','f'}
```

**Cabecera:**

SecuenciaCaracteres
.....
.....
+ void EliminaVarios(SecuenciaEnteros a_borrar)

**Llamada:**

```
int main(){  
    SecuenciaCaracteres secuencia;  
    SecuenciaEnteros posiciones_a_borrar;  
  
    secuencia.Aniade('a'); secuencia.Aniade('b'); secuencia.Aniade('c');  
    secuencia.Aniade('d'); secuencia.Aniade('e'); secuencia.Aniade('f');  
  
    posiciones_a_borrar.Aniade(1);  
    posiciones_a_borrar.Aniade(3);  
  
    secuencia.EliminaVarios(posiciones_a_borrar);  
}
```

**Implementación:**

Se deja como ejercicio a resolver.



**Un método puede recibir como parámetro un objeto de la MISMA clase**

**Ejemplo.** Queremos añadir todos los caracteres de una secuencia a otra.

```
secuencia --> {'a','b','c','d','e','f'}  
caracteres_a_anadir --> {'g','h'}  
secuencia --> {'a','b','c','d','e','f','g','h'}
```

**Cabecera:**

SecuenciaCaracteres
.....
.....
+ void AniadeVarios(SecuenciaCaracteres nuevos)

**Llamada:**

```
int main(){  
    SecuenciaCaracteres secuencia;  
    SecuenciaCaracteres caracteres_a_anadir;  
  
    secuencia.Aniade('a'); secuencia.Aniade('b'); secuencia.Aniade('c');  
    secuencia.Aniade('d'); secuencia.Aniade('e'); secuencia.Aniade('f');  
  
    caracteres_a_anadir.Aniade('g');  
    caracteres_a_anadir.Aniade('h');  
  
    secuencia.AniadeVarios(caracteres_a_anadir);  
    .....
```

## Implementación:

```
class SecuenciaCaracteres{
    .....
    void AniadeVarios(SecuenciaCaracteres nuevos){
        int totales_a_anadir = nuevos.TotalUtilizados();

        for (int i = 0; i < totales_a_anadir; i++)
            Aniade(nuevos.Elemento(i));          // Es importante entender
    }                                             // esta línea
};
```

<http://decsai.ugr.es/jccubero/FP/SecuenciaCaracteres.cpp>

## V.2.3. Objetos locales a un método y métodos que devuelven objetos

**Dentro de un método podemos declarar objetos**

- ▷ El objeto se creará en la pila, dentro del marco del método.
- ▷ Como cualquier otra variable local (automática), existirá mientras se esté ejecutando el método. Cuando termine la ejecución del método, automáticamente se libera la memoria asociada al objeto

```
class UnaClase{
    .....
};
class OtraClase{
    .....
    int UnMetodo(){
        UnaClase objeto;
        .....
    }
};
```

## Un método puede devolver un objeto

- ▷ Se declara el objeto local al método. El método devuelve una copia de dicho objeto local, a través de la sentencia `return`
- ▷ En el `main` (o desde dónde se llame al método) podremos asignar el objeto devuelto de dos formas:
  1. A través del operador de asignación (siempre que esté disponible).
  2. A través del constructor de copia.

```
class OtraClase{
    .....
    UnaClase Metodo(){
        UnaClase objeto_local;
        .....
        return objeto_local;
    }
};

int main(){
    UnaClase recibe_copia;
    OtraClase objeto;

    recibe_copia = objeto.Metodo();           // operador asignación

    UnaClase recibe_copia (objeto.Metodo()); // constructor de copia
```

**Nota.** Al ejecutar la sentencia `return` también se produce una llamada al constructor de copia, ya que es necesario crear un objeto temporal con los datos del objeto local, para así poder realizar la devolución del objeto.

**Ejemplo.** Añadimos un método a la clase `Recta` para que obtenga el `SegmentoDirigido` incluido en la recta delimitado por dos abscisas.

### Cabecera:

Recta	
	.....
+ <code>SegmentoDirigido</code>	<code>Segmento(</code> <div style="text-align: right;"><code>double abscisa_origen, double abscisa_final)</code></div>

### Llamada:

```
int main(){
    Recta una_recta(2, 3, -11); // 2x + 3y - 11 = 0

    // Error de compilación:
    // SegmentoDirigido no tiene constructor sin parámetros

    // SegmentoDirigido segmento;
    // segmento = una_recta.Segmento(3.5, 4.6);
```

El segmento no tiene un constructor sin parámetros. Por tanto, tenemos que recurrir a construir el segmento a partir de la recta, y asignarlo utilizando el constructor de copia:

```
int main(){
    Recta una_recta(2, 3, -11); // 2x + 3y - 11 = 0

    SegmentoDirigido segmento(una_recta.Segmento(3.5, 4.6)); // Const.copia

    cout << "\nLongitud segmento = " << segmento.Longitud();
```

### Implementación:

```
bool SonIguales(double uno, double otro) {
    return fabs(uno-otro) <= 1e-6;
}

class SegmentoDirigido{
    .....
};

class Recta{
    .....
    SegmentoDirigido Segmento(double abscisa_origen, double abscisa_final){
        double ordenada_origen = Ordenada_en(abscisa_origen);
        double ordenada_final  = Ordenada_en(abscisa_final);

        SegmentoDirigido segmento(abscisa_origen, ordenada_origen,
                                   abscisa_final, ordenada_final);

        return segmento;
    }
};
```

[http://decsai.ugr.es/jccubero/FP/V\\_punto\\_recta\\_segmento.cpp](http://decsai.ugr.es/jccubero/FP/V_punto_recta_segmento.cpp)

**A lo largo de esta asignatura, sólo se trabaja con copias de objetos. En particular, los métodos siempre devuelven copias de objetos.**

**IMPORTANT**

Un método de una clase  $C$  puede devolver un objeto de la MISMA clase  $C$

```
class MiClase{
    .....
    MiClase Metodo(){
        MiClase objeto;
        .....
        return objeto;
    }
};
```

**Ejemplo.** Dada la recta  $Ax + By + C = 0$ , la ecuación de un recta perpendicular a ésta pasando por el punto  $(x_0, y_0)$  es  $y - mx + mx_0 - y_0 = 0$ , dónde  $m$  es la pendiente de la nueva recta e igual al opuesto de la inversa de la pendiente de la otra recta, es decir:  $m = B/A$ .

### Cabecera:

Recta
.....
+ Recta Perpendicular(Punto2D pasando_por_aqui)

### Llamada:

```
int main(){
    Punto2D un_punto(5, 3);
    Recta una_recta(2, 3, -11);    // 2x + 3y - 11 = 0
    Recta perpendicular_a_una_recta(una_recta.Perpendicular(un_punto));
    cout << perpendicular_a_una_recta.Pendiente();
}
```

### Implementación:

```
class Recta{
    .....
    Recta Perpendicular (Punto2D pasando_por_aqui){
        double coef_y = 1;
        double pendiente_nueva = B/A;
        double coef_x = -pendiente_nueva;
        double coef_ind = pasando_por_aqui.Abscisa() * pendiente_nueva
                           - pasando_por_aqui.Ordenada();
        Recta perpendicular(coef_x, coef_y, coef_ind);
        return perpendicular;
    }
};
```

[http://decsai.ugr.es/jccubero/FP/V\\_punto\\_recta\\_segmento.cpp](http://decsai.ugr.es/jccubero/FP/V_punto_recta_segmento.cpp)



**Recordemos que también podemos declarar un vector local clásico dentro de un método:**

```
class MiClase{
    int Metodo(){
        int vector_local [TAMANIO];
        .....
        return entero;
    }
}
```

**pero recuerde:**

**Un método NO puede devolver un vector clásico**

```
class MiClase{
    int [] Metodo(){                // Error de compilación
        int vector [TAMANIO];
        .....
        return vector;
    }
}
```

**Lo que sí podemos hacer es devolver un objeto conteniendo un vector.**

**Un método o función no puede devolver un vector clásico  
(ver página 624).**

**Sin embargo, un método sí puede devolver un objeto que  
contenga como dato miembro un vector clásico.**

**IMPORTANT**

## Un método SÍ puede devolver un objeto conteniendo un vector clásico

**Ejemplo.** Añadimos un método a la clase `SecuenciaCaracteres` que devuelva una secuencia de enteros con las posiciones en las que se encuentra un carácter dado. En el `main`, llame al método `EliminaVarios` de la página 616 para eliminar todas las ocurrencias del carácter.

### Cabecera:

SecuenciaCaracteres	
.....	
+ <b>SecuenciaEnteros</b>	<code>PosicionesDe(char a_buscar)</code>

### Llamada:

```
int main(){
    SecuenciaCaracteres secuencia;
    SecuenciaEnteros posiciones_encontrado;
    char buscado;
    .....
    posiciones_encontrado = secuencia.PosicionesDe(buscado);
    secuencia.EliminaVarios(posiciones_encontrado);
}
```

**También podríamos haber usado el constructor de copia:**

```
int main(){
    SecuenciaCaracteres secuencia;
    .....
    SecuenciaEnteros posiciones_encontrado(secuencia.PosicionesDe(buscado));
    secuencia.EliminaVarios(posiciones_encontrado);
}
```

### Implementación:

```
class SecuenciaCaracteres{
    .....
    SecuenciaEnteros PosicionesDe(char a_buscar){
        SecuenciaEnteros posiciones_encontrado;

        for (int i = 0; i < total_utilizados; i++){
            if (vector_privado[i] == a_buscar)
                posiciones_encontrado.Aniade(i);
        }

        return posiciones_encontrado;
    }
};
```

**Ejemplo.** Construimos una clase para la lectura de secuencias de caracteres. Este sería un ejemplo de una clase cuya responsabilidad es *fabricar* objetos de otra clase (ver página 694)

**Cabecera:**

LectorSecuenciaCaracteres
- char terminador
+ LectorSecuenciaCaracteres(char caracter_terminador)
+ SecuenciaCaracteres Lee()

**Llamada:**

```
int main(){
    const char TERMINADOR = '#';
    LectorSecuenciaCaracteres lector_secuencias(TERMINADOR);
    SecuenciaCaracteres secuencia;
    SecuenciaCaracteres otra_secuencia;

    secuencia = lector_secuencias.Lee();
    otra_secuencia = lector_secuencias.Lee();
    .....
```

## Implementación:

```
class LectorSecuenciaCaracteres{
private:
    char terminador;
public:
    LectorSecuenciaCaracteres(char caracter_terminador)
        :terminador(caracter_terminador)
    {
    }

    SecuenciaCaracteres Lee(){
        SecuenciaCaracteres a_leer;
        int total_introducidos, capacidad_secuencia;
        char caracter;

        total_introducidos = 0;
        capacidad_secuencia = a_leer.Capacidad();
        caracter = cin.get();

        while (caracter == '\n') // Se salta los new line
            caracter = cin.get();

        while (caracter != terminador &&
            total_introducidos < capacidad_secuencia){
            a_leer.Aniade(caracter);
            total_introducidos++;
            caracter = cin.get();
        }

        return a_leer;
    }
};
```

<http://decsai.ugr.es/jccubero/FP/SecuenciaCaracteres.cpp>

**Ejemplo.** Sobre la clase `SecuenciaCaracteres`, añadimos un método que devuelva una nueva secuencia de caracteres, con los caracteres del actual pasados a mayúsculas:

**Cabecera:**

<code>SecuenciaCaracteres</code>
.....
+ <code>SecuenciaCaracteres</code> <code>ToUpper()</code>

**Llamada:**

```
int main(){
    SecuenciaCaracteres secuencia, secuencia_mayuscula;
    .....
    secuencia_mayuscula = secuencia.ToUpper();
}
```

**Implementación:**

```
class SecuenciaCaracteres{
    .....
    SecuenciaCaracteres ToUpper(){
        SecuenciaCaracteres en_mayuscula;

        for(int i = 0; i < total_utilizados; i++)
            en_mayuscula.Aniade(toupper(vector_privado[i]));

        return en_mayuscula;
    }
};
```

<http://decsai.ugr.es/jccubero/FP/SecuenciaCaracteres.cpp>

## V.2.4. Operaciones binarias entre objetos de una misma clase

¿Cómo diseñamos los métodos que trabajan sobre dos objetos de una misma clase?

**Las operaciones binarias que involucren dos objetos de una misma clase, usualmente se implementarán como métodos de dicha clase. El método actuará sobre un objeto y se le pasará como parámetro el otro objeto.**

**IMPORTANT**

### Ejemplos:

```
int pos_contiene = secuencia.PosicionContiene_a(pequenia);
bool se_intersecan = circunferencia.Interseca_con(otra_circunferencia);
son_iguales = un_punto.EsIgual_a(otro_punto);
Conjunto cjto_union = un_conjunto.Union_con(otro_conjunto);
Fraccion suma_fracciones(una_fraccion.Sumale(otra_fraccion));
```

---

### Nota:

Usualmente omitiremos los sufijos "le", "\_con", "\_a", etc. Por ejemplo:

Definiremos `una_fraccion.Suma(...)` en vez de  
`una_fraccion.Sumale(...)`

Definiremos `coordenadaGPS.Distancea(otra_coordenada)` en vez de  
`coordenadaGPS.Distancea_con(otra_coordenada)`

---

A veces, está claro qué objeto debemos fijar.

**Ejemplo.** Defina un método para ver si una secuencia de caracteres contiene a otra. El algoritmo lo vimos en la página 319. Ahora lo encapsulamos en un método.

**Cabecera:**

SecuenciaCaracteres
..... + int PosicionContiene(SecuenciaCaracteres a_buscar)

**Llamada:**

```
int main(){
    .....
    pos_contiene = secuencia.PosicionContiene(pequenia);

    if (pos_contiene == -1)
        cout << "\nNo encontrada";
    else
        cout << "\nEncontrada en la posición " << pos_contiene;
    .....
```

**Lo que nunca haremos será definir el método Contiene pasándole como parámetro dos secuencias:**

```
SecuenciaCaracteres esta_no_pinta_nada;
SecuenciaCaracteres secuencia, pequena;
.....
pos_contiene = esta_no_pinta_nada.Contiene(secuencia, pequena);
```





### **Implementación:**

```
int PosicionContiene (SecuenciaCaracteres a_buscar){
    int  inicio, posicion_contiene, ultima_componente;
    bool encontrado, va_coincidiendo;
    int utilizados_a_buscar = a_buscar.TotalUtilizados();

    if (utilizados_a_buscar > 0){
        ultima_componente = total_utilizados - utilizados_a_buscar;
        encontrado = false;

        for (inicio = 0; inicio <= ultima_componente && !encontrado;
              inicio++){
            va_coincidiendo = true;

            for (int i = 0; i < utilizados_a_buscar && va_coincidiendo; i++)
                va_coincidiendo = vector_privado[inicio + i]
                    == a_buscar.Elemento(i);
            if (va_coincidiendo){
                posicion_contiene = inicio;
                encontrado = true;
            }
        }
    }
    else
        encontrado = false;

    if (encontrado)
        return posicion_contiene;
    else
        return -1;
}
```

<http://decsai.ugr.es/jccubero/FP/SecuenciaCaracteres.cpp>

En otras ocasiones los dos objetos intervienen con idéntica *importancia*. En este caso, se fija cualquiera de ellos y se pasa como parámetro el otro.

**Ejercicio.** Añadimos a la clase `Fraccion` un método para sumarle otra fracción.

```
Fraccion una_fraccion(4, 9), otra_fraccion(5,2);
Fraccion suma_vs1(una_fraccion.Suma(otra_fraccion));
Fraccion suma_vs2(otra_fraccion.Suma(una_fraccion));
```

### Cabecera:

Fraccion	
- int	numerador
- int	denominador
+	Fraccion (int el_numerador, int el_denominador)
+ void	SetNumerador (int el_numerador)
+ void	SetDenominador (int el_denominador)
+ int	Numerador()
+ int	Denominador()
+ void	Reduce()
+ double	Division()
+ Fraccion	Suma(Fraccion otra_fraccion)

### Llamada:

```
int main(){
    int una_fraccion_numerador, una_fraccion_denominador,
        otra_fraccion_numerador, otra_fraccion_denominador;

    cout << "\nIntroduzca numerador y denominador de las dos fracciones ";
    cin >> una_fraccion_numerador;
    cin >> una_fraccion_denominador;
    cin >> otra_fraccion_numerador;
    cin >> otra_fraccion_denominador;
```

```
Fraccion una_fraccion (una_fraccion_numerador,  
                        una_fraccion_denominador);  
Fraccion otra_fraccion(otra_fraccion_numerador,  
                       otra_fraccion_denominador);  
  
Fraccion suma_fracciones (una_fraccion.Suma(otra_fraccion));  
suma_fracciones.Reduce();  
  
cout << "\nResultado de la suma: "  
      << suma_fracciones.Numerador() << " / "  
      << suma_fracciones.Denominador();
```

**Volvemos a repetir que nunca haremos el siguiente diseño:**

```
Fraccion esta_no_pinta_nada(3, 5);  
Fraccion una_fraccion(4, 9), otra_fraccion(5,2);  
Fraccion suma  
    (esta_no_pinta_nada.Suma(una_fraccion, otra_fraccion));
```



## Implementación:

```
class Fraccion{
    .....
    Fraccion Suma(Fraccion otra_fraccion){
        int suma_numerador;
        int suma_denominador;

        suma_numerador = numerador * otra_fraccion.Denominador() +
                        denominador * otra_fraccion.Numerador() ;
        suma_denominador = denominador * otra_fraccion.Denominador();

        Fraccion suma(suma_numerador, suma_denominador);
        suma.Reduce();

        return suma;
    }
};
```

[http://decsai.ugr.es/jccubero/FP/V\\_fraccion.cpp](http://decsai.ugr.es/jccubero/FP/V_fraccion.cpp)

Otro error común es definir una clase específica para realizar operaciones con dos fracciones:

```
class ParejaFracciones{
private:
    Fraccion una_fraccion;
    Fraccion otra_fraccion;
public:
    ParejaFracciones (Fraccion fraccion_primera,
                      Fraccion fraccion_segunda)
        :una_fraccion(fraccion_primera),
        otra_fraccion(fraccion_segunda)
    { }
    Fraccion Suma(){
        int suma_numerador, denominador_suma;
        suma_numerador = fraccion_primera.Numerador() *
                          fraccion_segunda.Denominador() +
                          fraccion_primera.Denominador() *
                          fraccion_segunda.Numerador() ;
        suma_denominador = fraccion_primera.Denominador() *
                            otra_fraccion.Denominador();
        Fraccion suma(suma_numerador, suma_denominador);
        suma.Reduce();
        return suma;
    }
};
```



Si tuviésemos que definir una clase `Pareja_C` para cada clase `C` sobre la que quisiésemos realizar operaciones con pares de objetos, el código se complicaría innecesariamente.

Como ya dijimos en la página **611**, tampoco definiremos funciones globales que actúen sobre objetos:

```
#include <iostream>
```

```
// Clase
```

```
class Fraccion{
```

```
    .....
```

```
};
```

```
// Función
```

```
Fraccion Suma (Fraccion una_fraccion, Fraccion otra_fraccion){
```

```
    .....
```

```
}
```



### **Ejemplo.** Comprobar si dos puntos son iguales.

Con una función global sería:

```
bool SonIguales (double un_real, double otro_real){  
    return abs(un_real-otro_real) <= 1e-6;  
}  
  
bool SonIgualesPuntos(Punto2D un_punto, Punto2D otro_punto){  
    return SonIguales(un_punto.Abscisa(), otro_punto.Abscisa())  
        && SonIguales(un_punto.Ordenada(), otro_punto.Ordenada());  
}
```



Definiendo métodos dentro de las clases:

#### **Cabecera:**

Punto2D
..... + double EsIgual_a(Punto2D otro_punto)

#### **Llamada:**

```
int main(){  
    bool son_iguales;  
    Punto2D un_punto(5.1, 3.2);  
    Punto2D otro_punto(5.1, 3.5);  
  
    son_iguales = un_punto.EsIgual_a (otro_punto);  
}
```

### Implementación:

```
bool SonIguales (double un_real, double otro_real){  
    return fabs(un_real-otro_real) <= 1e-6;  
}  
  
class Punto2D{  
    .....  
  
    bool EsIgual_a (Punto2D otro_punto){  
        return (SonIguales(abscisa, otro_punto.Abscisa()) &&  
                SonIguales(ordenada, otro_punto.Ordenada()));  
    }  
};
```



[http://decsai.ugr.es/jccubero/FP/V\\_punto\\_recta\\_segmento.cpp](http://decsai.ugr.es/jccubero/FP/V_punto_recta_segmento.cpp)

**Posteriormente se amplía esta discusión funciones vs clases.**



## V.2.5. Funciones globales, métodos privados y métodos públicos

Hemos visto que las funciones pueden definirse:

- ▷ Encapsuladas dentro de una clase (métodos)
- ▷ Globales, fuera de las clases

Como norma general, no usaremos funciones globales.

Sin embargo, en ocasiones, puede justificarse el uso de éstas:

- ▷ cuando sean funciones auxiliares genéricas que podamos utilizar en muchas clases diferentes,
- ▷ cuando sean funciones definidas para trabajar con tipos básicos predefinidos.

## ► Usando un método privado 😊

```
class Punto2D{
private:
    double abscisa;
    double ordenada;
    bool SonIguales(double uno, double otro)  {
        return fabs(uno-otro) <= 1e-6;
    }
public:
    Punto2D(double abscisaPunto, double ordenadaPunto)
        :abscisa(abscisaPunto),
        ordenada(ordenadaPunto)
    { }
    double Abscisa(){
        return abscisa;
    }
    double Ordenada(){
        return ordenada;
    }
    bool EsIgual_a (Punto2D otro_punto){
        return  SonIguales(abscisa, otro_punto.Abscisa()) &&
                SonIguales(ordenada, otro_punto.Ordenada());
    }
};
```

**Esta aproximación es correcta, pero ¿y si necesitamos comparar dos doubles en otras clases? Habría que repetir el código del método SonIguales en cada una de estas clases.**

## ► Usando una función global 😊

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

bool SonIguales(double uno, double otro) {
    return abs(uno-otro) <= 1e-6;
}

class Punto2D{
private:
    double abscisa;
    double ordenada;
public:
    Punto2D(double abscisaPunto, double ordenadaPunto)
        :abscisa(abscisaPunto),
        ordenada(ordenadaPunto)
    { }
    double Abscisa(){
        return abscisa;
    }
    double Ordenada(){
        return ordenada;
    }
    bool EsIgual_a (Punto2D otro_punto){
        return SonIguales(abscisa, otro_punto.Abscisa()) &&
            SonIguales(ordenada, otro_punto.Ordenada());
    }
};

int main(){
    <Aquí también puedo usar directamente SonIguales>
}
```

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

bool SonIguales(double uno, double otro) {
    return fabs(uno-otro) <= 1e-6;
}

class TransaccionBancaria{
public:
    double Importe(){
        .....
    }
    .....
};

int main(){
    TransaccionBancaria una_transaccion, otra_transaccion;
    bool son_iguales;
    .....
    son_iguales = SonIguales(una_transaccion.Importe(),
                             otra_transaccion.Importe());
    .....
}
```

---

### Otros ejemplos de funciones que podríamos poner globales:

```
int MaximoComunDivisor (int un_entero, int otro_entero)
double Maximo (double un_real, double otro_real)
char Transforma_a_Mayuscula (char un_caracter)
```

## ► Usando un método público 😞

¿Nos vale poner el método `SonIguales` como **público** y así poder usarlo en otros sitios? El compilador obviamente no lo puede impedir, pero es un diseño nefasto.

```
class Punto2D{
private:
    double abscisa;
    double ordenada;
public:
    bool SonIguales(double uno, double otro)  {
        return fabs(uno-otro) <= 1e-6;
    }
    .....
};

int main(){
    Punto2D que_pinta_aqui_un_punto(4.3, 5.8);
    bool son_iguales;
    double salario_A, salario_B;
    .....
    son_iguales =
        que_pinta_aqui_un_punto.SonIguales(salario_A, salario_B);
```



**Ejemplo.** Recuperemos el ejemplo del método `ToUpper` de una `SecuenciaCaracteres` (página 629)

```
class SecuenciaCaracteres{
    .....
    SecuenciaCaracteres ToUpper(){
        SecuenciaCaracteres en_mayuscula;

        for(int i = 0; i < total_utilizados; i++)
            en_mayuscula.Aniade(toupper(vector_privado[i]));

        return en_mayuscula;
    }
};
```

**Para transformar un carácter en su mayúscula, hemos usado la función `toupper` de la biblioteca `cctype`. ¿Y si usamos una función o método propio `ToMayuscula`?**

```
class SecuenciaCaracteres{
    .....
    SecuenciaCaracteres ToUpper(){
        SecuenciaCaracteres en_mayuscula;

        for(int i = 0; i < total_utilizados; i++)
            en_mayuscula.Aniade(ToMayuscula(vector_privado[i]));

        return en_mayuscula;
    }
};
```

**La cabecera sería:** `char ToMayuscula(char caracter)`. **Alternativas:**

- ▷ Definirlo como una función global: perfecto.
- ▷ Definirlo como un método privado: perfecto.
- ▷ Definirlo como un método publico: nefasto.

**En este caso, en el `main` o en cualquier otro sitio fuera de la clase, tendríamos:**

```
int main(){
    SecuenciaCaracteres no_pinta_nada;
    char letra, letra_mayuscula;
    .....
    letra_mayuscula = no_pinta_nada.ToUpper(letra);
}
```



**Cuando diseñe la interfaz pública de una clase, asegúrese que las llamadas a los métodos desde fuera de la clase son coherentes y representan acciones realizadas sobre un objeto:**

`objeto.MetodoQueOperaSobreElObjeto(...)`



## V.2.6. Acceso a los datos miembros privados de otros objetos

Si a un método de una clase le pasamos como parámetro (o declaramos local) un objeto *de la misma clase*, podremos acceder a sus datos miembros *privados* usando la notación con punto.

Es coherente que ésto se permita con objetos de la misma clase. De hecho, la mayor parte de los lenguajes orientados a objetos lo permiten.

**Ejemplo.** Implementamos el método `SonIguales` de la clase `Punto2D` de otra forma alternativa:

```
class Punto2D{
    .....
    bool EsIgual_a (Punto2D otro_punto){
        return (SonIguales(abscisa, otro_punto.Abscisa()) &&
                SonIguales(ordenada, otro_punto.Ordenada()));
    }
};

class Punto2D{
    .....
    bool EsIgual_a (Punto2D otro_punto){
        return (SonIguales(abscisa, otro_punto.abscisa) &&
                SonIguales(ordenada, otro_punto.ordenada));
    }
};
```



**Ejemplo.** Implementamos el método `Suma` de la clase `Fraccion` de otra forma alternativa:

```
class Fraccion{
    .....
    Fraccion Suma(Fraccion otra_fraccion){
        int suma_numerador;
        int suma_denominador;

        suma_numerador = numerador * otra_fraccion.Denominador() +
                        denominador * otra_fraccion.Numerador() ;
        suma_denominador = denominador * otra_fraccion.Denominador();

        Fraccion suma(suma_numerador, suma_denominador);
        suma.Reduce();

        return suma;
    } };
```

```
class Fraccion{
    .....
    Fraccion Suma(Fraccion otra_fraccion){
        int suma_numerador;
        int suma_denominador;

        suma_numerador = numerador * otra_fraccion.denominador +
                        denominador * otra_fraccion.numerador ;
        suma_denominador = denominador * otra_fraccion.denominador;

        Fraccion suma(suma_numerador, suma_denominador);
        suma.Reduce();

        return suma;
    } };
```

**Ejemplo.** Implementamos el método `AniadeVarios` de la clase `SecuenciaCaracteres` de otra forma alternativa:


```
class SecuenciaCaracteres{
    .....
    void AniadeVarios(SecuenciaCaracteres nuevos){
        int totales_a_aniadir = nuevos.TotalUtilizados();

        for (int i = 0; i < totales_a_aniadir; i++)
            Aniade(nuevos.Elemento(i));
    }
};

class SecuenciaCaracteres{
    .....
    void AniadeVarios(SecuenciaCaracteres nuevos){
        int totales_a_aniadir = nuevos.TotalUtilizados();


        for (int i = 0; i < totales_a_aniadir; i++)
            Aniade(nuevos.vector_privado[i]);
    }
};
```

Si tenemos un objeto local de la misma clase, podemos acceder y **modificar** sus datos miembros privados, aunque usualmente lo evitaremos y será más seguro hacerlo a través de los métodos.

```
class SecuenciaCaracteres{
    .....
    SecuenciaCaracteres ToUpper(){
        SecuenciaCaracteres en_mayuscula;

        for(int i = 0; i < total_utilizados; i++){
            en_mayuscula.Aniade(toupper(vector_privado[i]));

        return en_mayuscula;
    }
};

class SecuenciaCaracteres{{
    .....
    SecuenciaCaracteres ToUpper(){
        SecuenciaCaracteres en_mayuscula;

        for(int i = 0; i < total_utilizados; i++){
            en_mayuscula.vector_privado[i] = toupper(vector_privado[i]);
            en_mayuscula.total_utilizados++; // Que no se olvide!
        }

        return en_mayuscula;
    }
};
```

Está claro que la primera versión es preferible ya que la tarea de añadir (asignar e incrementar el total de utilizados) es responsabilidad del método `Aniade`.

***Puede acceder a los datos miembros privados de otros objetos de la misma clase pero reserve esta posibilidad, en la medida de lo posible, para aquellos casos en los que sólo necesite consultar su valor. Por contra, fomente el acceso a los datos miembro de otros objetos de la misma clase a través de los métodos definidos en dichos objetos.***

## V.3. Objetos como datos miembro de otros objetos

### V.3.1. Objetos simples

¿Un objeto puede ser dato miembro de otro objeto? Por supuesto.

- ▷ Como cualquier otro dato miembro, C++ permite que sea privado o público. Nosotros siempre usaremos datos privados.
- ▷ Su uso sigue las mismas normas de programación que cualquier otro dato miembro. Sólo usaremos objetos como datos miembro, cuando esté claro que forman parte del *núcleo* de la clase.
- ▷ La clase del objeto contenido debe declararse antes que la clase del objeto contenedor.
- ▷ La clase contenedora y la clase contenida no pueden ser la misma (con punteros sí se puede)
- ▷ En el objeto de la clase contenedora se crea una instancia de la clase contenida.

```
class ClaseContenida{  
    <datos miembro y métodos>  
};  
  
class ClaseContenedora{  
    <private o public>:  
        ClaseContenida dato_miembro;  
};
```

**¿Qué ocurre cuando el constructor de la clase contenida tiene parámetros? Hay que crear el objeto con los parámetros adecuados. ¿Dónde? Obligatoriamente, en la lista de inicialización del constructor.**

```
class ClaseContenida{
public:
    ClaseContenida(int parametro){
        .....
    }
    .....
};

class ClaseContenedora{
private:
    // ClaseContenida dato_miembro(5); Error compilación
    ClaseContenida dato_miembro;
public:
    ClaseContenedora()
        :dato_miembro(5) // Llamada al constructor con parámetros
    { ..... }          // de ClaseContenida
    .....
};
```

**O en general:**

```
class ClaseContenedora{
private:
    ClaseContenida dato_miembro;
public:
    ClaseContenedora(int valor_inicial)
        :dato_miembro(valor_inicial) // Llamada al
                                        // constructor con parámetros
    { ..... }                        // de ClaseContenida
    .....
};
```

### **Ejemplo. Segmento.**

**Redefinimos la clase `SegmentoDirigido` para que contenga dos puntos como datos miembro en vez de 4 `double`.**

**Al constructor del segmento le pasamos 4 `double` (las coordenadas) para poder construir los puntos.**

Punto2D	
- double	abscisa
- double	ordenada
+	Punto2D(double abscisaPunto, double ordenadaPunto)
+ void	SetCoordenadas (double abscisaPunto, double ordenadaPunto)
+ double	Abscisa()
+ double	Ordenada()
+ bool	EsIgual_a(Punto2D otro_punto)

SegmentoDirigido	
- Punto2D	origen
- Punto2D	final
+	SegmentoDirigido(double origen_abscisa, double origen_ordenada, double final_abscisa, double final_ordenada)
+ void	SetCoordenadas (double origen_abscisa, double origen_ordenada, double final_abscisa, double final_ordenada)
+ Punto2D	Origen()
+ Punto2D	Final()
+ double	Longitud()
+ void	TrasladaHorizontal(double unidades)
+ void	TrasladaVertical(double unidades)
+ void	Traslada(double en_horizontal, double en_vertical)

```
class Punto2D{
    <no cambia nada>
};

class SegmentoDirigido{
private:
    Punto2D origen;
    Punto2D final;
public:
    // Constructor SegmentoDirigido con parámetros de tipo double
    SegmentoDirigido(double origen_abscisa, double origen_ordenada,
                     double final_abscisa, double final_ordenada)
        // Llamada al constructor de los puntos con parámetros
        :origen (origen_abscisa, origen_ordenada),
        final  (final_abscisa, final_ordenada)
    {
    }
    Punto2D Origen(){
        return origen;
    }
    Punto2D Final(){
        return final;
    }
    double Longitud(){
        double sumando_abscisa = origen.Abscisa()-final.Abscisa();
        double sumando_ordenada = origen.Ordenada()-final.Ordenada();

        return sqrt(sumando_abscisa * sumando_abscisa      +
                     sumando_ordenada * sumando_ordenada);
    }
    void TrasladaHorizontal(double unidades){
        origen.SetCoordenadas( origen.Abscisa() + unidades,
                               origen.Ordenada());
    }
}
```



```
        final.SetCoordenadas( final.Abscisa() + unidades,
                               final.Ordenada());
    }
    void TrasladaVertical(double unidades){
        origen.SetCoordenadas( origen.Abscisa(),
                               origen.Ordenada() + unidades);
        final.SetCoordenadas( final.Abscisa(),
                               final.Ordenada() + unidades);
    }
    void Traslada(double en_horizontal, double en_vertical){
        TrasladaHorizontal(en_horizontal);
        TrasladaVertical(en_vertical);
    }
};

int main(){
    SegmentoDirigido segmento (3.4, 4.5, 6.7, 9.2);

    Punto2D copia_de_origen (segmento.Origen());

    cout << copia_de_origen.Abscisa();    // 3.4
    cout << copia_de_origen.Ordenada();   // 4.5
```

**¿Qué ocurre si cambiamos las coordenadas de copia\_de\_origen?**

```
copia_de_origen.SetCoordenadas (9.1, 10.2);
```

**Obviamente, no se modifican las coordenadas del origen del segmento.**

El anterior ejemplo pone de manifiesto lo siguiente (es una ampliación de lo visto en la página 657):

**A lo largo de esta asignatura, sólo se trabaja con copias de objetos. En particular, los métodos siempre devuelven copias de objetos.**

**Por lo tanto, si un objeto  $A$  tiene como dato miembro un objeto  $B$  y un método de  $A$  devuelve una copia de  $B$ , las modificaciones que se hagan sobre dicha copia no afectan a  $B$ .**

**IMPORTANT**

En vez de construir el objeto *interno* en la clase contenedora, también podríamos construir el objeto fuera y pasarlo como parámetro al constructor. En dicho caso, debemos inicializar el dato miembro en la lista de inicialización, asignándole una copia del objeto pasado como parámetro (el compilador invoca al constructor de copia del dato miembro)

```
class ClaseContenida{
public:
    ClaseContenida(int parametro){
        .....
    }
    <datos miembro y métodos>
};

class ClaseContenedora{
private:
    ClaseContenida dato_miembro;
public:
    ClaseContenedora(ClaseContenida objeto)
        :dato_miembro(objeto) // Llamada al constructor de copia
    { ..... }                // de ClaseContenida
    .....
};
```

**Ejemplo. Segmento. Constructor con objetos como parámetros.**

SegmentoDirigido	
- Punto2D	origen
- Punto2D	final
+	SegmentoDirigido(Punto2D punto_origen, Punto2D punto_final)
+ void	SetCoordenadas (Punto2D punto_origen, Punto2D punto_final)
+ Punto2D	Origen()
+ Punto2D	Final()
+ double	Longitud()
+ void	TrasladaHorizontal(double unidades)
+ void	TrasladaVertical(double unidades)
+ void	Traslada(double en_horizontal, double en_vertical)

**Al constructor del segmento le pasamos directamente dos objetos de la clase Punto2D, en vez de los 4 double de las coordenadas.**

```
class Punto2D{
    <no cambia nada>
};

class SegmentoDirigido{
private:
    Punto2D origen;
    Punto2D final;
public:
    // Constructor SegmentoDirigido con parámetros de tipo Punto2D
    SegmentoDirigido(Punto2D punto_origen, Punto2D punto_final)
        :origen (punto_origen), // llamada al constructor de copia del punto
        final  (punto_final)    // llamada al constructor de copia del punto
    {
    }
    .....
};
```

```
int main(){
    Punto2D origen(3.4, 4.5);
    Punto2D final(6.7, 9.2);
    SegmentoDirigido segmento (origen, final);

    cout << segmento.Origien().Abscisa();    // 3.4
    cout << segmento.Origien().Ordenada();  // 4.5
```

---

**Ejemplo.** Si usamos esta versión del segmento, el método `Segmento` de la clase `Recta` visto en la página 621 que devolvía el segmento delimitado por dos valores de abscisas habría que reescribirlo como sigue:

```
class Punto2D{
    .....
};
class Recta{
private:
    .....
public:
    .....
    SegmentoDirigido Segmento(double abscisa_origen, double abscisa_final){
        double ordenada_origen = Ordenada_en(abscisa_origen);
        double ordenada_final  = Ordenada_en(abscisa_final);

        Punto2D pto_origen(abscisa_origen, ordenada_origen);
        Punto2D pto_final(abscisa_final, ordenada_final);

        SegmentoDirigido segmento(pto_origen, pto_final);

        return segmento;
    }
};
```

---

### **Ampliación:**



Supongamos que queremos comprobar que los datos a asignar en el constructor son correctos. Si no lo son, ¿qué podemos hacer?

- ▷ Lo ideal sería lanzar una excepción en el constructor, en la misma lista de inicialización del constructor, lo que impediría que se construyesen los datos miembros y por supuesto el objeto.
- ▷ Como no sabemos excepciones, tenemos que dejar que se construyan los datos miembros (en el ejemplo anterior son los dos objetos **Punto2D**) en la lista de inicialización del constructor. Una vez finalizada ésta, ya dentro del conjunto de sentencias del constructor, cambiaríamos los datos miembros y pondríamos algún valor especial (en el caso de que los parámetros no fuesen correctos).

Así se hace en el siguiente enlace:

---

[http://decsai.ugr.es/jccubero/FP/V\\_punto\\_recta\\_segmento\\_con\\_puntos\\_datos\\_miembro.cpp](http://decsai.ugr.es/jccubero/FP/V_punto_recta_segmento_con_puntos_datos_miembro.cpp)

**Ejemplo.** Añadimos a la clase CuentaBancaria un dato miembro de tipo Fecha para representar la fecha de apertura.

CuentaBancaria	
- double	saldo
- double	identificador
- Fecha	fecha_apertura
- bool	EsCorrectoSaldo(double saldo_propuesto)
- bool	EsCorrectoIdentificador(string identificador_propuesto)
- void	SetSaldo(double saldo_propuesto)
- void	SetIdentificador(string identificador_cuenta)
+	CuentaBancaria(string identificador_cuenta, double saldo_inicial, Fecha fecha_apertura_cuenta)
+ string	Identificador()
+ double	Saldo()
+ void	Ingresa(double cantidad)
+ void	Retira(double cantidad)
+ void	AplicaInteresPorcentual(int tanto_porcentaje)

```
class CuentaBancaria{
private:
    Fecha fecha_apertura;
    double saldo;
    const string identificador;
public:
    CuentaBancaria(string identificador_cuenta, double saldo_inicial,
                   Fecha fecha_apertura_cuenta)
        :saldo(saldo_inicial),
          identificador(identificador_cuenta),
          fecha_apertura(fecha_apertura_cuenta) // constr. de copia de Fecha
    { }
    .....
    Fecha FechaApertura(){
        return fecha_apertura;
    }
}
```

```
};
```

```
int main(){  
    Fecha una_fecha(12, 11, 2015);  
    CuentaBancaria cuenta("2031450100001367", 2000,  una_fecha);  
    .....  
}
```



Si se quiere inicializar un objeto dato miembro a unos valores por defecto, se puede hacer de dos formas:

- ▷ O bien usando la inicialización de los datos miembros entre llaves (disponible en C++ 11):

```
class SegmentoDirigido{
private:
    Punto2D origen = {0.0, 0.0};
    Punto2D final  = {1.0, 1.0};
    .....
}
```

- ▷ O bien usando en la lista de inicialización del constructor un *objeto sin nombre (unnamed object)* :

```
class SegmentoDirigido{
private:
    Punto2D origen;
    Punto2D final;
public:
    SegmentoDirigido()
        :origen(Punto2D(0.0, 0.0)), final(Punto2D(1.0, 1.0))
    {}
    .....
}
```

## V.3.2. Vectores de objetos

Recordemos que para declarar un vector de objetos de una clase  $B$ , ésta debe proporcionar un constructor sin parámetros (ver página 609):

```
class Punto2D{
    .....
    Punto2D(double abscisaPunto, double ordenadaPunto){
        .....
    }
};

int main(){
    const int TAMANIO = 20;
    Punto2D un_punto; // Error de compilación.
                        // Punto2D no tiene un constructor sin parámetros.

    Punto2D vector_de_puntos[TAMANIO]; // Error de compilación
```

---

```
class Punto2D{
    .....
    Punto2D(){
    }
    Punto2D(double abscisaPunto, double ordenadaPunto){
        .....
    }
};

int main(){
    const int TAMANIO = 20;
    Punto2D un_punto; // Correcto
    Punto2D vector_de_puntos[TAMANIO]; // Correcto. Se crean 20 puntos,
```

Lo mismo ocurre si dentro de una clase *A* tenemos como dato miembro un vector de objetos de otra clase *B*:

- ▷ Si la clase *B* proporciona un constructor sin parámetros, podremos declarar el vector y se crearán automáticamente los objetos.

```
class Punto2D{
    .....
    Punto2D(){
    }
    Punto2D(double abscisaPunto, double ordenadaPunto){
        .....
    }
};

class ConjuntoPuntos2D{
private:
    static const int TAMANIO = 20;
    Punto2D vector_de_puntos[TAMANIO];
    // Correcto. Se crean 20 puntos,
};
```

---

```
class Punto2D{
    .....
    Punto2D(double abscisaPunto, double ordenadaPunto){
        .....
    }
};

class ConjuntoPuntos2D{
private:
    static const int TAMANIO = 20;
    Punto2D vector_de_puntos[TAMANIO];    // Error de compilación
    .....
};
```

- ▷ En caso contrario, habría que crear todos los objetos en la lista de inicialización del constructor de *A* (no veremos cómo hacerlo).

**Ejemplo.** Construir una clase para albergar una colección de cuentas bancarias.

ColeccionCuentasBancarias	
- const int	NUM_CUENTAS
- CuentaBancaria	cjto_cuentas [NUM_CUENTAS]
- int	utilizados
+ void	Aniade(CuentaBancaria cuenta)
+ int	NumeroCuentas()
+ int	BuscaIndice(string identificador_a_buscar)
+ double	Saldo(int indice_cuenta)
+ string	Identificador(int indice_cuenta)
+ void	Ingresa(int indice_cuenta, double cantidad)
+ void	Retira(int indice_cuenta, double cantidad)
+ void	AplicaInteresPorcentual(int indice_cuenta, int tanto_porcentaje)

¿Incluimos en la colección un método del tipo?

```
CuentaBancaria Cuenta(int indice_cuenta)
```

Este método devolvería una copia de una cuenta. La manipulación (ingresos, retiradas, etc) de dicha copia no afectaría a la componente original de la colección (página 621). Por eso, preferimos no incluir dicho método.

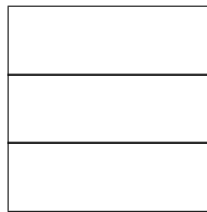
En el segundo cuatrimestre aprenderá a acceder directamente a las componentes de un vector dato miembro.

[http://decsai.ugr.es/jccubero/FP/V\\_cuenta\\_bancaria\\_fecha.cpp](http://decsai.ugr.es/jccubero/FP/V_cuenta_bancaria_fecha.cpp)

## V.4. Tablas de datos

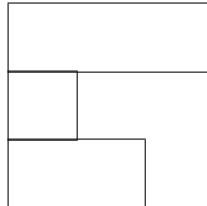
De forma genérica, diremos que una tabla es un conjunto de filas. Distinguiremos las siguientes situaciones:

- ▷ **Tabla rectangular**: todas las filas tienen el mismo número de columnas.




`Imagen2D`, `DatosPersonalesAlumnos`, `RegistrosPluviometricos`, **etc.**

- ▷ **Tabla dentada**: el número de columnas en cada fila no tiene por qué ser el mismo.




`ItemsPedidoCliente`, `ViviendasEnPropiedad`, **etc.**

Definiremos clases para representar ambas situaciones. Ahora bien, para cualquiera de ellas, podremos usar dos implementaciones alternativas:

- ▷ O bien implementaremos la clase usando matrices (doble corchete) de tipos simples.
- ▷ O bien implementaremos la clase usando vectores de objetos.

Las dos implementaciones son válidas. Dependiendo del tipo de operaciones que vayamos a realizar, éstas serán más fáciles de programar con una implementación u otra.

## V.4.1. Tablas rectangulares

Queremos representar una tabla de datos con la siguiente ocupación:


### V.4.1.1. Representación de una tabla rectangular usando una matriz

En la página 348 vimos cómo hacerlo usando una matriz de caracteres:

X	X	X	?	?	?	...	?
X	X	X	?	?	?	...	?
?	?	?	?	?	?	...	?
...	...	...	...	...	...	...	...
?	?	?	?	?	?	...	?

**Ahora vamos a ver cómo encapsular las operaciones dentro de una clase:**

- ▷ **Todas las filas tendrán el mismo número de columnas. ¿Cuándo y cómo lo imponemos?**
  - **Lo mejor es especificarlo en el constructor.**
  - **O bien pasamos al constructor un entero (`num_columnas`)**
  - **O bien pasamos al constructor la primera fila (y de dicha fila obtenemos el número de columnas).**

**Por simplificar, impondremos como precondition que el número de columnas especificado en el constructor está en el rango correcto.**

- ▷ **Si vamos a ocupar un bloque completo, lo lógico es no permitir añadir caracteres uno a uno sino una fila completa. Así, la tabla estará siempre en un estado válido. Lo conseguimos definiendo el método:**

```
void Aniade(SecuenciaCaracteres fila_nueva)
```

- ▷ **Podremos recuperar o bien el carácter de una casilla concreta:**

```
char Elemento (int fila, int columna)
```

**o bien una fila entera:**

```
SecuenciaCaracteres Fila(int indice)
```

**Observe que internamente usamos una matriz de corchetes, pero en los métodos pasamos y devolvemos objetos de la clase `SecuenciaCaracteres`**

- ▷ **Si queremos buscar la posición de una componente, devolveremos un struct:**

```
struct ParFilaColumna{  
    int fila;  
    int columna;  
};
```

## Ejemplo. Sopa de letras

```

A  C  E  R  O  S  ?  ...  ?
L  A  S  E  R  I  ?  ...  ?
T  I  O  B  D  E  ?  ...  ?
A  D  I  N  E  D  ?  ...  ?
R  A  Z  O  N  P  ?  ...  ?
?  ?  ?  ?  ?  ?  ?  ...  ?

...

?  ?  ?  ?  ?  ?  ?  ...  ?

```

ParFilaColumna
+ int fila
+ int columna

SopaLetras	
- const int	<u>MAX_FIL</u>
- const int	<u>MAX_COL</u>
- const int	util_col
- char	matriz_privada[MAX_FIL] [MAX_COL]
- int	util_fil
+	SopaLetras(int numero_de_columnas)
+	SopaLetras(SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas()
+ int	FilasUtilizadas()
+ int	ColUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ ParFilaColumna	BuscaPalabra (SecuenciaCaracteres a_buscar)



```
int main(){
    const char TERMINADOR = '#';
    char letra;
    int  numero_columnas, longitud_a_buscar;
    ParFilaColumna encontrado;
    SecuenciaCaracteres cadena,  a_buscar;

    // Formato entrada: número columnas y caracteres de la sopa por filas
    // El número de caracteres debe ser múltiplo del número de columnas

    cin >> numero_columnas;
    SopaLetras sopa(numero_columnas);

    cin >> letra;

    while (letra != TERMINADOR){
        for (int i = 0; i < numero_columnas; i++){
            cadena.Aniade(letra);
            cin >> letra;
        }
        sopa.Aniade(cadena);
        cadena.EliminaTodos();
    }
    cin >> longitud_a_buscar;

    for (int i = 0; i < longitud_a_buscar; i++){
        cin >> letra;
        a_buscar.Aniade(letra);
    }
    encontrado = sopa.BuscaPalabra(a_buscar);
    cout << encontrado.fila << " " << encontrado.columna;
}
```

**El único método que no es trivial es `BuscaPalabra` y el algoritmo que lo implementaba ya se vio en la página 355**

```
struct ParFilaColumna{
    int fila;
    int columna;
};

class SecuenciaCaracteres{
    .....
};

class SopaLetras{
private:
    static const int MAX_FIL = 50;
    static const int MAX_COL = 40;
    char matriz_privada[MAX_FIL][MAX_COL];
    int util_fil;
    const int util_col;
public:
    // Prec: 0 < numero_de_columnas <= MAX_COL(40)
    SopaLetras(int numero_de_columnas)
        :util_fil(0), util_col(numero_de_columnas)
    {
    }
    // Prec: primera_fila.TotalUtilizados() <= MAX_COL(40)
    SopaLetras(SecuenciaCaracteres primera_fila)
        :util_fil(0), util_col(primer_fila.TotalUtilizados())
    {
        Anade(primer_fila);
    }
    int CapacidadFilas(){
        return MAX_FIL;
    }
    int FilasUtilizadas(){
        return util_fil;
    }
};
```

```
}
int ColUtilizadas(){
    return util_col;
}
char Elemento(int fila, int columna){
    return matriz_privada[fila][columna];
}
SecuenciaCaracteres Fila(int indice_fila){
    SecuenciaCaracteres fila;

    for (int col = 0; col < util_col; col++)
        fila.Aniade(matriz_privada[indice_fila][col]);

    return fila;
}
void Aniade(SecuenciaCaracteres fila_nueva){
    int numero_columnas_nueva;

    if (util_fil < MAX_FIL){
        numero_columnas_nueva = fila_nueva.TotalUtilizados();

        if (numero_columnas_nueva == util_col){
            for (int col = 0; col < util_col ; col++)
                matriz_privada[util_fil][col] = fila_nueva.Elemento(col);
            util_fil++;
        }
    }
}
ParFilaColumna BuscaPalabra (SecuenciaCaracteres a_buscar){
    bool encontrado, va_coincidiendo;
    ParFilaColumna pos_encontrado;
    int tamano_a_buscar;
```

```
    encontrado = false;
    pos_encontrado.fila = pos_encontrado.columna = -1;
    tamaño_a_buscar = a_buscar.TotalUtilizados();

    if (tamaño_a_buscar <= util_col){
        for (int fil = 0; fil < util_fil && !encontrado; fil++){

            for (int col_inicio = 0;
                col_inicio + tamaño_a_buscar <= util_col && !encontrado ;
                col_inicio++){

                va_coincidiendo = true;

                for (int i = 0; i < tamaño_a_buscar && va_coincidiendo; i++)
                    va_coincidiendo = matriz_privada[fil][col_inicio + i]
                        ==
                        a_buscar.Elemento(i);

                if (va_coincidiendo){
                    encontrado = true;
                    pos_encontrado.fila = fil;
                    pos_encontrado.columna = col_inicio;
                }
            }
        }
    }

    return pos_encontrado;
}

};
```

[http://decsai.ugr.es/jccubero/FP/V\\_sopa.cpp](http://decsai.ugr.es/jccubero/FP/V_sopa.cpp)

## ¿Qué pasaría al ejecutar el siguiente código?

```
cadena = sopa.Fila(1);  
cadena.Aniade('o');           // Se modifica cadena pero NO la sopa
```

***Si desde un objeto `obj` llamamos a un método que devuelve un objeto `dev`, las modificaciones que hagamos sobre `dev` no afectan a `obj`.***

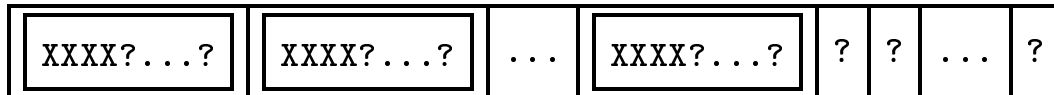
**En esta asignatura estamos viendo cómo manejar objetos en C++, siempre a través de copias de éstos (al pasarlos como parámetros, al devolverlos en un método, etc)**

**Otra alternativa es trabajar con referencias a objetos y no con copias. Esta forma de trabajar se consigue en C++ usando *referencias (references)* y *punteros (pointers)* y se verá en el segundo cuatrimestre.**

**IMPORTANT**

### V.4.1.2. Representación de una tabla rectangular usando un vector de objetos

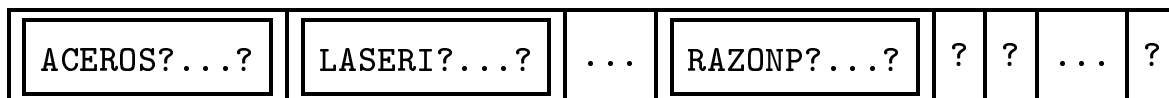
Queremos representar una tabla rectangular como un vector de secuencias, es decir, un vector en el que cada componente es un objeto de una clase `Secuencia`.



El máximo número de columnas viene determinado ahora por la capacidad de la clase `Secuencia`.

**Ejemplo.** Implementamos la sopa de letras usando internamente un vector de `SecuenciaCaracteres`.

A	C	E	R	O	S	?	...	?
L	A	S	E	R	I	?	...	?
T	I	O	B	D	E	?	...	?
A	D	I	N	E	D	?	...	?
R	A	Z	O	N	P	?	...	?
?	?	?	?	?	?	?	...	?
...								
?	?	?	?	?	?	?	...	?



**La interfaz pública no varía.**

**Lo que cambia son los datos privados:**

SopaLetras	
- const int	MAX_FIL
- const int	util_col
- <b>SecuenciaCaracteres</b>	vector_privado[MAX_FIL]
- int	util_fil
+	SopaLetras(int numero_de_columnas)
+	SopaLetras(SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas()
+ int	FilasUtilizadas()
+ int	ColUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ ParFilaColumna	BuscaPalabra (SecuenciaCaracteres a_buscar)

**Como la interfaz pública no ha cambiado, la función `main` es la misma (página 672). Al igual que pasaba en la primera versión:**

```
cadena = sopa.Fila(1);  
cadena.Aniade('o');           // Se modifica cadena pero NO la sopa
```

***Si un método devuelve una copia de una componente de un vector dato miembro, las modificaciones que se realizan posteriormente sobre la copia no afectan a la componente original.***

## Implementación:

Vemos primero los métodos más básicos.

```
class SopaLetras{
private:
    static const int MAX_FIL = 50;
    SecuenciaCaracteres vector_privado[MAX_FIL];
    int util_fil = 0;
    const int util_col;
public:
    // Prec: 0 < numero_de_columnas <= Capacidad de SecuenciaCaracteres
    SopaLetras(int numero_de_columnas)
        :util_col(numero_de_columnas)
    {
    }
    // No hay ninguna precondition
    SopaLetras(SecuenciaCaracteres primera_fila)
        :util_col(primera_fila.TotalUtilizados())
    {
        Aniade(primera_fila);
    }
    // 0 bien:
    // SopaLetras(SecuenciaCaracteres primera_fila)
    //      :SopaLetras(primera_fila.TotalUtilizados())
    // {
    //      Aniade(primera_fila);
    // }

    int CapacidadFilas(){
        return MAX_FIL;
    }
    int FilasUtilizadas(){
        return util_fil;
    }
}
```



```
int ColUtilizadas(){
    return util_col;
}
char Elemento(int fila, int columna){
    return vector_privado[fila].Elemento(columna);
}
SecuenciaCaracteres Fila(int indice_fila){
    return vector_privado[indice_fila];
}
void Aniade(SecuenciaCaracteres  fila_nueva){
    int numero_columnas_nueva;

    if (util_fil < MAX_FIL){
        numero_columnas_nueva = fila_nueva.TotalUtilizados();

        if (numero_columnas_nueva == util_col){
            vector_privado[util_fil] = fila_nueva;
            util_fil++;
        }
    }
}
ParFilaColumna BuscaPalabra (SecuenciaCaracteres a_buscar){
    .....
}
};
```

## ¿Cómo implementamos el método BuscaPalabra?

- ▷ De forma similar a como se hizo en la versión que usaba una matriz de doble corchete.
- ▷ Mejor aún. ¿No es lógico reutilizar el método PosicionContiene de la clase SecuenciaCaracteres visto en la página 631?

SecuenciaCaracteres
..... + int PosicionContiene(SecuenciaCaracteres pequena)

**Cambiamos por tanto la implementación del método BuscaPalabra de la clase SopaLetras para que llame al método PosicionContiene de la clase SecuenciaCaracteres**

```
class SopaLetras{
.....
ParFilaColumna BuscaPalabra (SecuenciaCaracteres a_buscar){
    bool encontrado;
    ParFilaColumna pos_encontrado;
    int tamano_a_buscar;
    SecuenciaCaracteres fila;
    int col_encontrado;

    encontrado = false;
    pos_encontrado.fila = pos_encontrado.columna = -1;
    tamano_a_buscar = a_buscar.TotalUtilizados();

    if (tamano_a_buscar <= util_col){
        for (int i = 0; i < util_fil && !encontrado; i++){
            fila = vector_privado[i];
            col_encontrado = fila.PosicionContiene(a_buscar);

            if (col_encontrado != -1){
```

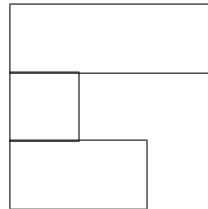
```
        encontrado = true;
        pos_encontrado.fila = i;
        pos_encontrado.columna = col_encontrado;
    }
}
return pos_encontrado;
};
```

[http://decsai.ugr.es/jccubero/FP/V\\_sopa\\_vector\\_objetos.cpp](http://decsai.ugr.es/jccubero/FP/V_sopa_vector_objetos.cpp)

***Fomente la creación de métodos genéricos situándolos en aquellas clases que previsiblemente vayan a reutilizarse en más ocasiones.***

## V.4.2. Tablas dentadas

Queremos representar una tabla de datos con la siguiente ocupación:



### V.4.2.1. Representación de una tabla dentada usando una matriz

En la página 358 vimos cómo hacerlo usando una matriz de caracteres:

X	X	X	X	X	?	...	?
X	X	?	?	?	?	...	?
X	X	X	X	?	?	...	?
?	?	?	?	?	?	...	?
...	...	...	...	...	...	...	...
?	?	?	?	?	?	...	?

Ahora vamos a ver cómo encapsular las operaciones dentro de una clase:

- ▷ Cada fila tendrá un número de columnas distinto. Lo controlamos con un vector `util_col`  
 Por simplificar, impondremos como precondition que el número de columnas especificado en el constructor está en el rango correcto.
- ▷ Al igual que con las tablas rectangulares, sólo se permitirán añadir filas enteras y podremos recuperar o bien el carácter de una casilla concreta o bien una fila entera.

**Ejemplo.** Podemos considerar un objeto de la clase `Texto` como un conjunto de líneas. Internamente, cada línea corresponde a una fila de una matriz de caracteres.

```

l  í  n  e  a  s  ?  ?  ?  ...  ?
d  e  ?  ?  ?  ?  ?  ?  ?  ...  ?
d  i  s  t  i  n  t  o  ?  ...  ?
t  a  m  a  ñ  o  ?  ?  ?  ...  ?
?  ?  ?  ?  ?  ?  ?  ...  ?  ...  ?
?  ?  ?  ?  ?  ?  ?  ...  ?  ...  ?
...
?  ?  ?  ?  ?  ?  ?  ...  ?  ...  ?

```

Texto	
- const int	<u>MAX_FIL</u>
- const int	<u>MAX_COL</u>
- char	matriz_privada[MAX_FIL][MAX_COL]
- int	util_col[MAX_FIL]
- int	util_fil
- void	ReemplazaFila (int fila_a_borrar, int fila_origen)
+	Texto()
+ int	CapacidadFilas()
+ int	FilasUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	Inserta(SecuenciaCaracteres fila_nueva, int pos_fila_insercion)

```
int main(){
    const char TERMINADOR_LINEA = '.';
    const char TERMINADOR_TEXTO = '#';
    char letra;
    SecuenciaCaracteres cadena, a_insertar;
    Texto texto;

    // Formato de entrada:
    // Primera fila.Segunda.Ultima fila.#

    cin >> letra;

    while (letra != TERMINADOR_TEXTO){
        while (letra != TERMINADOR_LINEA){
            cadena.Aniade(letra);
            cin >> letra;
        }
        texto.Aniade(cadena);
        cadena.EliminaTodos();
        cin >> letra;
    }

    cin >> letra;

    while (letra != TERMINADOR_LINEA){
        a_insertar.Aniade(letra);
        cin >> letra;
    }

    texto.Aniade(a_insertar);
    cadena = texto.Fila(texto.FilasUtilizadas()-1);
    cout << cadena.ToString();
}
```

Todos los métodos son inmediatos de implementar. Únicamente hay que ir controlando el número de columnas de cada fila.

El único método nuevo es el que inserta una fila entera. El algoritmo sería el siguiente:

**Algoritmo: Insertar una fila en una matriz**

```
Recorrer todas las filas desde el final hasta
la posición de inserción.
    Reemplazar cada fila por la anterior
Volcar la nueva fila en la posición de inserción.
```

```
class Texto{
private:
    static const int MAX_FIL = 50;
    static const int MAX_COL = 40;
    char matriz_privada[MAX_FIL][MAX_COL];
    int util_fil;
    int util_col[MAX_FIL];

    void ReemplazaFila (int fila_a_borrar, int fila_origen){
        int columnas_utilizadas = util_col[fila_origen];

        for (int col = 0; col < columnas_utilizadas; col++)
            matriz_privada[fila_a_borrar][col] =
                matriz_privada[fila_origen][col];

        util_col[fila_a_borrar] = columnas_utilizadas;
    }
public:
    // Prec: 0 < numero_de_columnas <= MAX_COL(40)
    Texto()
```

```
        :util_fil(0)
    {
        for (int i=0; i<MAX_COL; i++)
            util_col[i] = 0;
    }
    int CapacidadFilas(){
        return MAX_FIL;
    }
    int FilasUtilizadas(){
        return util_fil;
    }
    int ColUtilizadas(int indice_fila){
        return util_col[indice_fila];
    }
    char Elemento(int fila, int columna){
        return matriz_privada[fila][columna];
    }
    SecuenciaCaracteres Fila(int indice_fila){
        SecuenciaCaracteres fila;
        int num_columnas = util_col[indice_fila];

        for (int j = 0; j < num_columnas; j++)
            fila.Aniade(matriz_privada[indice_fila][j]);

        return fila;
    }

    void Aniade(SecuenciaCaracteres fila_nueva){
        int numero_columnas_nueva;

        if (util_fil < MAX_FIL){
            numero_columnas_nueva = fila_nueva.TotalUtilizados();
```



```
        if (numero_columnas_nueva < MAX_COL){
            for (int j = 0; j < numero_columnas_nueva ; j++)
                matriz_privada[util_fil][j] = fila_nueva.Elemento(j);

            util_col[util_fil] = numero_columnas_nueva;
            util_fil++;
        }
    }

void Inserta(SecuenciaCaracteres fila_nueva, int pos_fila_insercion){
    int numero_columnas_fila_nueva = fila_nueva.TotalUtilizados();

    if (numero_columnas_fila_nueva <= MAX_COL &&
        util_fil < MAX_FIL &&
        pos_fila_insercion >= 0 &&
        pos_fila_insercion <= util_fil){

        for (int i = util_fil ; i > pos_fila_insercion ; i--)
            ReemplazaFila(i, i-1);

        for (int j = 0; j < numero_columnas_fila_nueva; j++)
            matriz_privada[pos_fila_insercion][j] =
                fila_nueva.Elemento(j);

        util_fil++;
        util_col[pos_fila_insercion] = numero_columnas_fila_nueva;
    }
}

};
```

[http://decsai.ugr.es/jccubero/FP/V\\_texto.cpp](http://decsai.ugr.es/jccubero/FP/V_texto.cpp)

### V.4.2.2. Representación de una tabla dentada usando un vector de objetos

El tipo de estructura que vamos a construir es similar a la tabla rectangular con un vector de objetos, sólo que ahora no nos preocupamos de que todas las filas tengan el mismo número de columnas.

XX??????...	XXXXXX??...	...	XXX?????...	?	?	...	?
-------------	-------------	-----	-------------	---	---	-----	---

Cada fila será un objeto de la clase `SecuenciaCaracteres` por lo que la gestión de la longitud de cada una de ellas la haremos en la clase `SecuenciaCaracteres` y no en la clase `Texto`. Por tanto, ya no es necesario el dato miembro vector `util_col` de `Texto` (ver página 686)

```
fila = tabla.Fila(0);
columnas_usadas = fila.TotalUtilizados();
```

**Ejemplo.** Implementamos la clase `Texto` usando internamente un vector de `SecuenciaCaracteres`.

```

l  í  n  e  a  s  ?  ?  ?  ...  ?
d  e  ?  ?  ?  ?  ?  ?  ?  ...  ?
d  i  s  t  i  n  t  o  ?  ...  ?
t  a  m  a  ñ  o  ?  ?  ?  ...  ?
?  ?  ?  ?  ?  ?  ?  ...  ?  ...  ?
?  ?  ?  ?  ?  ?  ?  ...  ?  ...  ?

...

?  ?  ?  ?  ?  ?  ?  ...  ?  ...  ?
    
```

líneas?...?	de?...?	...	tamaño?...?	?	?	...	?
-------------	---------	-----	-------------	---	---	-----	---

**La interfaz pública no varía.**

**Lo que cambia son los datos privados:**

Texto	
- const int	MAX_FIL
- SecuenciaCaracteres	vector_privado[MAX_FIL]
- int	util_fil
<hr/>	
+	Texto()
+ int	CapacidadFilas()
+ int	FilasUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	Inserta(SecuenciaCaracteres fila_nueva, int pos_fila_insercion)

**Como la interfaz pública no ha cambiado, la función `main` es la misma (página 686). Al igual que pasaba en la primera versión:**

```
cadena = texto.Fila(1);  
cadena.Aniade('o');           // Se modifica cadena pero NO el texto
```

## Implementación:

```
class Texto{
private:
    static const int MAX_FIL = 50;
    SecuenciaCaracteres vector_privado[MAX_FIL];
    int util_fil;
public:
    // Prec: 0 < numero_de_columnas <= Capacidad de SecuenciaCaracteres
    Texto()
        :util_fil(0)
    {
    }
    int CapacidadFilas(){
        return MAX_FIL;
    }
    int FilasUtilizadas(){
        return util_fil;
    }
    int ColUtilizadas(int indice_fila){
        return vector_privado[indice_fila].TotalUtilizados();
    }
    char Elemento(int fila, int columna){
        return vector_privado[fila].Elemento(columna);
    }
    SecuenciaCaracteres Fila(int indice_fila){
        return vector_privado[indice_fila];
    }
    void Aniade(SecuenciaCaracteres fila_nueva){
        if (util_fil < MAX_FIL){
            vector_privado[util_fil] = fila_nueva;
            util_fil++;
        }
    }
}
```

```
}  
void Inserta(SecuenciaCaracteres fila_nueva, int pos_fila_insercion){  
    if (util_fil < MAX_FIL &&  
        pos_fila_insercion >= 0 &&  
        pos_fila_insercion <= util_fil){  
  
        for (int i = util_fil ; i > pos_fila_insercion ; i--)  
            vector_privado[i] = vector_privado[i-1];  
  
        vector_privado[pos_fila_insercion] = fila_nueva;  
        util_fil++;  
    }  
}  
};
```

[http://decsai.ugr.es/jccubero/FP/V\\_texto\\_vector\\_objetos.cpp](http://decsai.ugr.es/jccubero/FP/V_texto_vector_objetos.cpp)

## V.5. Temas de Ampliación

### V.5.1. Fábricas de objetos (Ampliación)

*Este apartado no entra en el examen*



Sabemos que no podemos mezclar en una misma clase cálculos con E/S. ¿Cómo encapsularíamos las tareas de leer o imprimir los datos miembro de una clase?

*Las tareas necesarias para realizar las operaciones de E/S de los datos de un objeto, se realizarán en clases específicas que implementen dichas responsabilidades.*

*Dada una clase  $C$ , será usual crear otra clase **fábrica (factory)** que construya objetos de  $C$  a través de un método (el método devolverá un objeto de  $C$ ).*

*Para mostrar los datos de  $C$  se puede crear otra clase a la que se le pasará como parámetro el objeto de  $C$  cuyos datos queramos mostrar.*

**Ejemplo.** Definimos clases para leer e imprimir puntos y rectas.

LectorRectas
- string mensaje
+ LectorRectas (string mensaje_entrada_datos) + void ImprimeMensajeEntrada() + <b>Recta</b> Lee()

ImpresorRectas
+ Imprime ( <b>Recta</b> una_recta)

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

```
class Punto2D{
    <no cambia nada>
};
class Recta{
    <no cambia nada>
};
```



```
class LectorRectas{
private:
    string mensaje;
public:
    LectorRectas(string mensaje_entrada_datos)
        :mensaje (mensaje_entrada_datos)
    { }
    void ImprimeMensajeEntrada(){
        cout << mensaje;
    }
    Recta Lee(){
        double A, B, C;
        cin >> A;
        cin >> B;
        cin >> C;

        Recta recta(A, B, C);
        return recta;
    }
};

class LectorPuntos{
private:
    string mensaje;
public:
    LectorPuntos(string mensaje_entrada_datos)
        :mensaje (mensaje_entrada_datos)
    { }
    void ImprimeMensajeEntrada(){
        cout << mensaje;
    }
}
```

```
Punto2D Lee(){
    double abscisa, ordenada;
    cin >> abscisa;
    cin >> ordenada;

    Punto2D punto(abscisa, ordenada);
    return punto;
}

};

class ImpresorPuntos{
public:
    void Imprime(Punto2D punto){
        cout << "(" << punto.Abscisa() << ","
              << punto.Ordenada() << ")";
    }
};

class ImpresorRectas{
public:
    void Imprime (Recta una_recta){
        cout << una_recta.CoeficienteA() << " x + " <<
              una_recta.CoeficienteB() << " y + " <<
              una_recta.CoeficienteC() << " = 0";
    }
};

int main(){
    bool contiene;
    LectorPuntos lector_de_puntos("\nIntroduzca las dos coordenadas
                                   del punto\n");
    LectorRectas lector_de_rectas("\nIntroduzca los tres coeficientes
                                   de la recta\n");

    ImpresorRectas impresor_de_rectas;
    ImpresorPuntos impresor_de_puntos;
```

```
lector_de_puntos.ImprimeMensajeEntrada();
Punto2D un_punto(lector_de_puntos.Lee()); // Constructor de copia
lector_de_rectas.ImprimeMensajeEntrada();
Recta una_recta(lector_de_rectas.Lee()); // Constructor de copia

contiene = una_recta.Contiene(un_punto);

cout << "\nLa recta ";
impresor_de_rectas.Imprime(una_recta);

if (contiene)
    cout << " contiene al punto ";
else
    cout << " no contiene al punto ";

impresor_de_puntos.Imprime(un_punto);
```

## V.5.2. Tratamiento de errores con excepciones (Ampliación)

*Esta sección no entra en el examen*



Supongamos que se violan las precondiciones de un método ¿Cómo lo notificamos al cliente del correspondiente objeto?

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    void Aniade(char nuevo){
        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
        else
            cout << "No hay componentes suficientes";
    }
};

int main(){
    SecuenciaCaracteres cadena;

    cadena.Aniade('h');
    .....
    cadena.Aniade('a'); // Si ya no cabe => "No hay componentes suficientes"
```



**Esta solución rompe la norma básica de no mezclar C-E/S**

El método `Aniade` ha sido el encargado de tratar el error producido por la violación de la precondition. Esto es poco **flexible** 😞

*Si se detecta un error en un método (como la violación de una precondición) la respuesta (acciones a realizar para tratar el error) debe realizarse fuera de dicho método*

La solución *clásica* es devolver un código de error (bool, char, entero, enumerado, etc)

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    int Aniade(char nuevo){
        int error = 0;

        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
        else
            error = 1;

        return error;
    }
};

int main(){
    SecuenciaCaracteres cadena;
```

```
.....  
if (cadena.Aniade('h') == 1)  
    cout << "No hay componentes suficientes";  
else{  
    <sigue la ejecución del programa>  
};
```

### Problema: El excesivo anidamiento oscurece el código:

```
int main(){  
    SecuenciaCaracteres cadena;  
  
    if (cadena.Aniade('h') == 1)  
        cout << "No hay componentes suficientes";  
    else{  
        if (cadena.Aniade('o') == 1)  
            cout << "No hay componentes suficientes";  
        else  
            if (cadena.Aniade('y') == 1)  
                cout << "No hay componentes suficientes";  
            else  
                <sigue la ejecución del programa>  
    };  
}
```

**Solución:** Utilizar el mecanismo de gestión de excepciones.

En PDO una **excepción** (*exception*) es un **objeto** que contiene información que es traspasada desde el sitio en el que ocurre un problema a otro sitio en el que se tratará dicho problema.

La idea es la siguiente:

- ▷ Un método **genera una excepción** con información sobre un error.  
La generación de la excepción se realiza a través del operador `throw`
- ▷ El flujo de control sale del método y va directamente a un lugar en el que se **gestiona el error**.  
Dicho lugar queda determinado por un bloque `try - catch`.

En C++, una excepción puede ser de cualquier tipo de dato: entero, real, nuestras propias clases, etc. Por ahora, utilizaremos clases de excepción estándar, incluidos en la biblioteca `stdexcept`, como por ejemplo:

- ▷ Clase `logic_error`:  
Se usa para notificar que ha habido un error *lógico* como por ejemplo la violación de una precondición de un método.
- ▷ Clase `runtime_error`:  
Se usa para notificar que ha habido un error *de ejecución* como por ejemplo una división entera entre cero o un acceso a un fichero inexistente.

Hay una sobrecarga de los constructores de dichas clases que acepta una cadena de caracteres. En dicha cadena indicaremos el error que se ha producido.

Una vez creado el objeto de excepción se consulta dicha cadena con el método `what()`.

```
#include <stdexcept>

class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    void Aniade(char nuevo){
        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
        else{
            logic_error objeto_excepcion("No hay componentes suficientes");
            throw objeto_excepcion;
        }
    }
};

int main(){
    SecuenciaCaracteres cadena;

    try{
        cadena.Aniade('h');
        .....
        cadena.Aniade('a');    // Si ya no cabe =>
                               // El flujo de control salta al catch
        .....
    }
    catch(logic_error excepcion){
        cout << "Error lógico: " << excepcion.what();
    }
}
```



Por ahora, el tratamiento del error dentro del `catch` únicamente consiste en informar con un `cout`.

Si no construimos el bloque `try catch` y salta una excepción, el programa abortará y perderemos el control de su ejecución.

```
int main(){
    SecuenciaCaracteres cadena;

    cadena.Aniade('h');
    .....
    cadena.Aniade('a'); // Si ya no cabe => El programa aborta
```

**Si se desea, las dos sentencias:**

```
logic_error objeto_excepcion("No hay componentes suficientes");  
throw objeto_excepcion;
```

**pueden resumirse en una sola:**

```
throw logic_error("No hay componentes suficientes");
```

**Quedaría así:**

```
#include <stdexcept>  
class SecuenciaCaracteres{  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados;  
public:  
    .....  
    void Aniade(char nuevo){  
        if (total_utilizados < TAMANIO){  
            vector_privado[total_utilizados] = nuevo;  
            total_utilizados++;  
        }  
        else  
            throw logic_error("No hay componentes suficientes");  
            // Se crea el objeto de la clase logic_error  
            // El flujo de control sale del método  
    }  
};
```

## El resto de métodos también pueden generar excepciones:

```
class SecuenciaCaracteres{
    .....
    void Inserta(int pos_insercion, char valor_nuevo){
        if ((total_utilizados < DIM) && (pos_insercion>=0)
            && (pos_insercion <= total_utilizados)){

            for (int i=total_utilizados ; i>pos_insercion ; i--)
                vector_privado[i] = vector_privado[i-1];

            vector_privado[pos_insercion] = valor_nuevo;
            total_utilizados++;
        }
        else
            throw logic_error("Posición de inserción inválida");
    }
};

int main(){
    SecuenciaCaracteres cadena;

    try{
        cadena.Aniade('h');
        .....
        cadena.Inserta(60, 'a');
        .....
    }
    catch(logic_error excepcion){
        cout << "Error lógico: " << excepcion.what();
    }
}
```

**Según sea la excepción que se lance, el método `what()` devolverá** Posición de inserción inválida **o bien** No hay componentes suficientes.

**En el mismo bloque try-catch podemos capturar excepciones generadas por métodos de distintas clases:**

```
class SecuenciaCaracteres{
    .....
};
class SopaLetras{
    .....
    void Aniade(SecuenciaCaracteres fila_nueva){
        if (util_fil < MAX_FIL){
            vector_privado[util_fil] = fila_nueva;
            util_fil++;
        }
        else
            throw logic_error("No hay más filas disponibles");
    }
};
int main(){
    SecuenciaCaracteres cadena;
    SopaLetras sopa;

    try{
        cadena.Aniade('h');
        .....
        sopa.Aniade(cadena);
        .....
    }
    catch(logic_error excepcion){
        cout << "Error lógico: " << excepcion.what();
    }
}
```

¿Y si se generan excepciones de tipos distintos? Hay que añadir el `catch` correspondiente.

```
class LectorSecuenciaCaracteres{
private:
    const char terminador;
    const string nombre_fichero;
    ifstream lector;                // Para leer de un fichero
public:
    LectorSecuenciaCaracteres(string nombre_fichero_entrada,
                               char terminador_datos)
        :terminador(terminador_datos),
          nombre_fichero(nombre_fichero_entrada)
    {
    }
    void AbreFichero(){
        lector.open(nombre_fichero);

        if (lector.fail()){
            string mensaje_error = "Error en la apertura del fichero " +
                                   nombre_fichero;
            throw runtime_error(mensaje_error);
        }
    }
    void CierraFichero(){....}
    SecuenciaCaracteres LeeSecuencia(){....}
    bool HayDatos(){....}
};

int main(){
    SecuenciaCaracteres cadena;
    SopaLetras sopa;
    const char terminador = '#';
    LectorSecuenciaCaracteres input_cadena("DatosEntrada.txt", terminador);
```

```
try{
    input_vector.AbreFichero();
    .....
    cadena = input_cadena.LeeSecuencia();
    .....
    sopa.Aniade(cadena);
    .....
}
catch(logic_error excepcion){
    cout << "Error lógico: " << excepcion.what();
}
catch(runtime_error excepcion){
    cout << "Error de ejecución: " << excepcion.what();
}
```

### A tener en cuenta:

- ▷ Si se produce una excepción de un tipo, el flujo de control entra en el `catch` que captura el correspondiente tipo. No entra en los otros bloques `catch`.
- ▷ Si se omite el `catch` correspondiente a un tipo y se produce una excepción de dicho tipo, el programa abortará.

***El tratamiento de errores de programación usando el mecanismo de las excepciones se basa en asociar el error con un TIPO de excepción***

Supongamos que un método A llama a otro B y éste a otro C. ¿Qué pasa si salta una excepción en C?

El flujo de control va saliendo de todas las llamadas de los métodos, hasta llegar a algún sitio en el que haya una sentencia `try-catch`. Este proceso se conoce como *stack unwinding* (limpieza de la pila sería una posible traducción). Lo recomendable es lanzar la excepción en los últimos niveles y capturarla en los primeros (`main`) → *Throw early catch late*

---

### Consideraciones finales:

- ▷ El uso de excepciones modifica el flujo de control del programa. Un mal programador podría forzar su uso en sustitución de estructuras de control básicas como las condicionales.

Siempre reservaremos el uso de excepciones para programar casos excepcionales, como por ejemplo, cuando la violación de las precondiciones de un método pueda tener efectos potencialmente peligrosos (recordar lo visto en la página 555)

- ▷ En C++, una excepción puede ser de cualquier tipo de dato: entero, real, nuestras propias clases, etc.

Al crear nuestras propias clases de excepción, podremos construir objetos con más información que un simple `string` sobre el error producido.

- ▷ Cuando se vea el concepto de *Herencia* se verá cómo pueden capturarse diferentes tipos de excepciones en un único `catch`. Para ello, tendremos varias clases de excepción que heredan de otra clase *base*.

## V.5.3. Ciclo de vida del software (Ampliación)

*Esta sección no entra en el examen*



Fases esenciales durante el desarrollo del software:

▷ **Planificación (planning)**

Comprende el **análisis de requisitos (requirements analysis)** para establecer las necesidades del cliente, así como el análisis de costes y riesgos.

▷ **Análisis (analysis) y diseño (design)**

En esta fase se analiza qué metodología de programación (procedural, funcional, orientada a objetos, etc) se adapta mejor a la resolución del problema y se diseña la arquitectura de la solución.

En el tema V se verán algunos conceptos relacionados con el diseño de una solución usando PDO.

También se evalúan los recursos hardware/software disponibles.

▷ **Implementación (Implementation)**

Una vez realizado el diseño, se procede a su implementación o **codificación (coding)**.

▷ **Validación (validation) y Verificación (verification)**

Durante la validación comprobamos que el software construido cumple los requerimientos del cliente (**hemos construido el producto adecuado**). En esta fase es necesario interactuar con el cliente.

Durante la verificación comprobamos que el software construido no tiene errores (**hemos construido adecuadamente el producto**). En esta fase no se interactúa con el cliente. Se desarrollan una serie o batería de **pruebas (tests)** para comprobar el correcto funcionamiento:

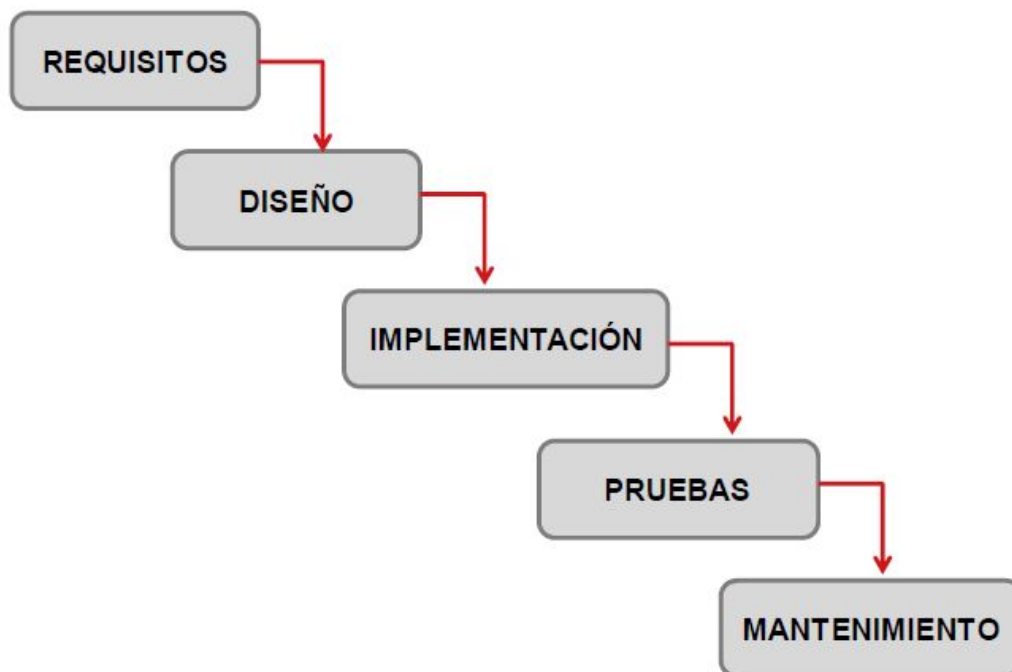


- **Pruebas de unidad (unit testing)** . Son pruebas dirigidas a comprobar el correcto funcionamiento de un módulo (por ejemplo, una función o un objeto)
- **Pruebas de integración (integration tests)** . Son pruebas dirigidas a comprobar la correcta integración entre los módulos desarrollados.

▷ **Desarrollo (deployment) y mantenimiento (maintenance)** .

Una vez construido el software, se procede a su distribución en un entorno de producción, generación de documentación, programas de aprendizaje, marketing, etc. Durante la explotación real, pueden cambiar o surgir nuevos requerimientos que necesiten volver al inicio del ciclo de vida.

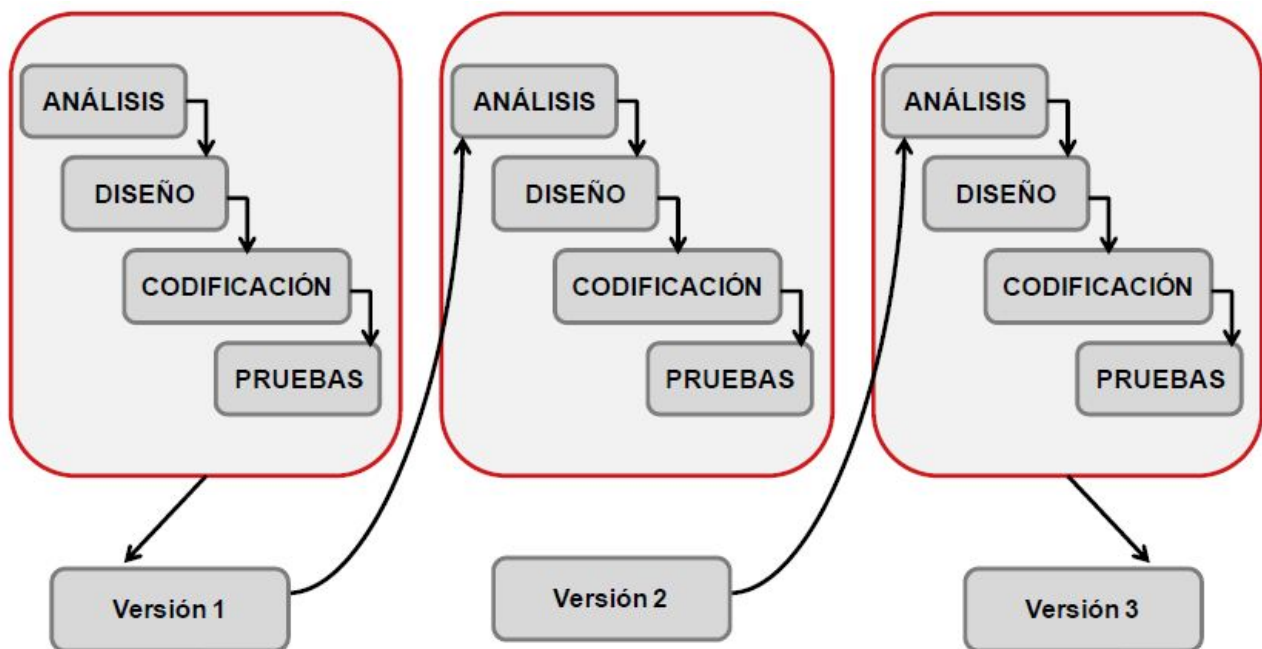
El ciclo de vida básico sería el **modelo en cascada (waterfall model)** :



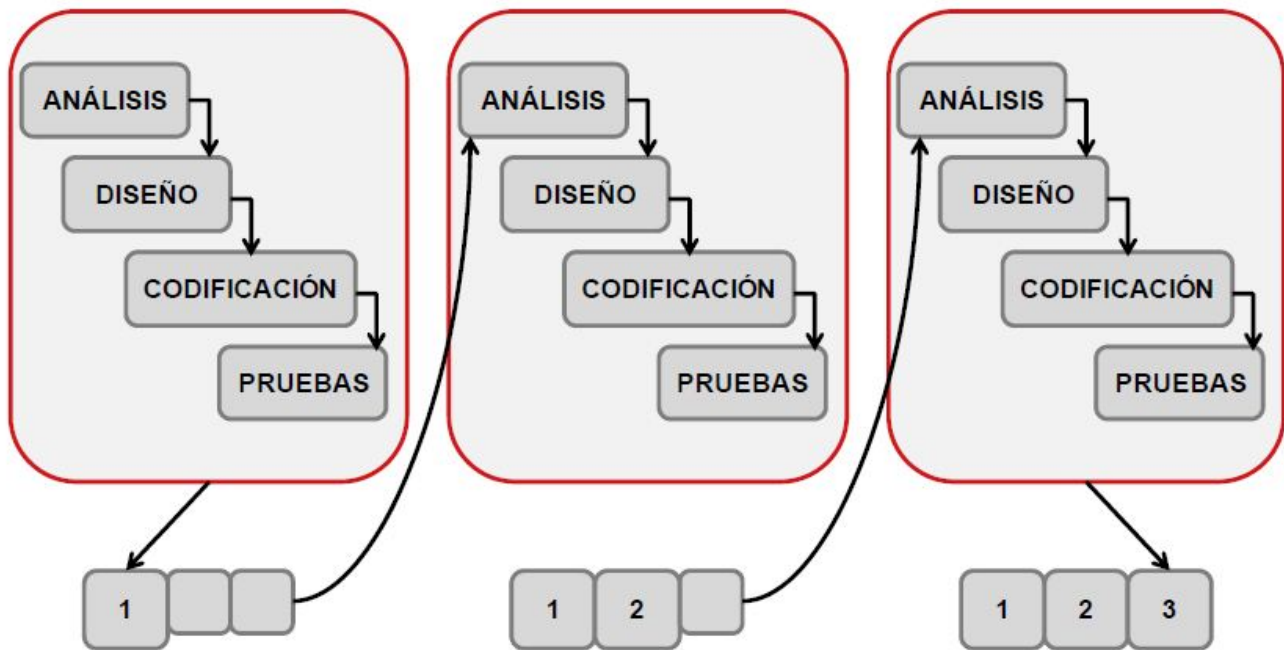
Sin embargo, en el desarrollo de cualquier proyecto software de mediana envergadura deben tenerse en cuenta otros factores como:

- ▷ **Análisis de riesgos**
- ▷ **Tiempo dedicado al desarrollo de cada fase**
- ▷ **Interacción entre los equipos de desarrollo**
- ▷ **Interacción con el cliente**
- ▷ **Secuenciación temporal entre las fases**
- ▷ **Necesidad de obtener resultados (prototipos) rápidamente**

La inclusión de estos factores lleva a considerar ciclos de vida más complejos como por ejemplo los *modelos iterativos (iterative models)* :



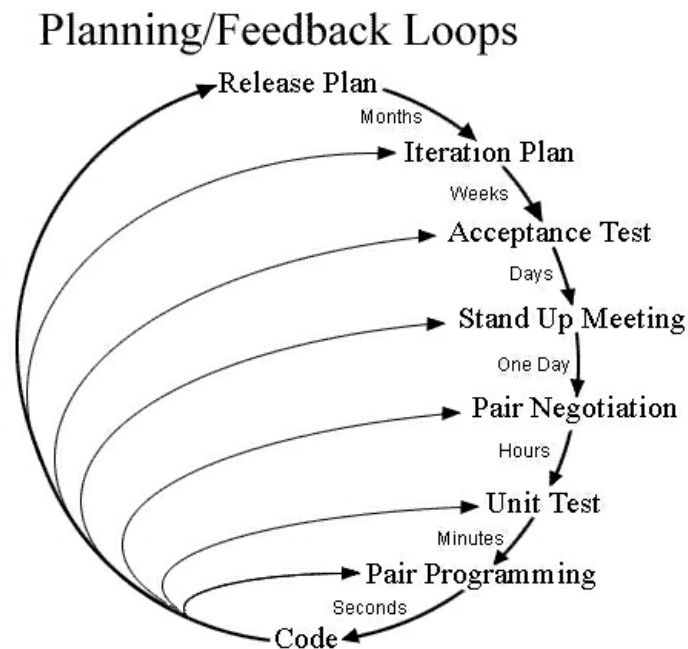
*modelos incrementales (incremental models) :*



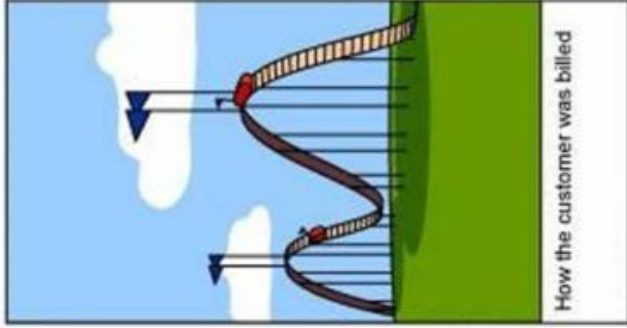
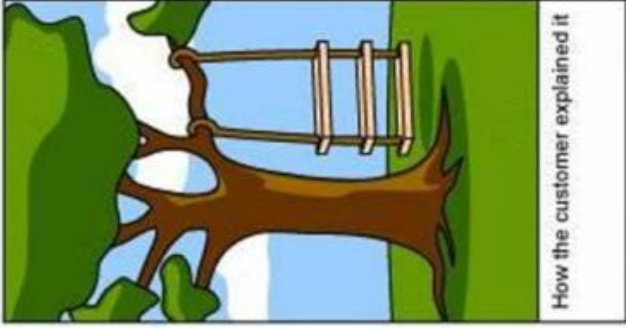
*modelos en espiral (spiral models) :*



*modelos de desarrollo ágil (agile development models)* como la *programación extrema (extreme programming)* :



Estos conceptos se analizan con detalle en la asignatura *Ingeniería de Software (Software Engineering)*



# Índice alfabético

- álgebra de boole (boolean algebra), 133
- ámbito (scope), 39, 258, 377, 398, 437
- ámbito público (public scope), 456
- ámbito privado (private scope), 456
- índice de variación, 37
- string, 593
- acceso aleatorio (random access), 295
- acceso directo (direct access), 295
- acceso fuera de rango (out of bound access), 301
- algoritmo (algorithm), 3
- análisis (analysis), 711
- análisis de requisitos (requirements analysis), 711
- ascii, 85
- asociatividad (associativity), 51
- búsqueda binaria (binary search), 313
- búsqueda secuencial (linear/sequential search), 313
- biblioteca (library), 14
- bit, 46
- bucle controlado por condición (condition-controlled loop), 210
- bucle controlado por contador (counter controlled loop), 210
- bucle post-test (post-test loop), 210
- bucle pre-test (pre-test loop), 210
- buffer, 103
- byte, 46
- código binario (binary code), 6
- código fuente (source code), 9
- cabecera (header), 363
- cadena de caracteres (string), 81
- cadena vacía (empty string), 312, 595
- clase (class), 434
- code point, 87
- codificación (coding), 87, 711
- cohesión (cohesion), 518
- coma flotante (floating point), 57
- compilación separada (separate compilation), 543
- compilador (compiler), 13
- componentes (elements/components), 293
- componentes léxicos (tokens), 19



comportamiento (behaviour), 432, 435

comportamiento indeterminado (undefined behaviour), 301, 375, 423

condición (condition), 116

conjunto de caracteres (character set), 85

constante (constant), 33

constantes a nivel de clase (class constants), 538

constantes a nivel de objeto (object constants), 538

constantes estáticas (static constants), 538

constructor (constructor), 481

constructor de copia (copy constructor), 605

constructor de oficio, 488

constructor por defecto (default constructor), 488

cursor (cursor), 103

dato (data), 15

datos globales (global data), 398

datos locales (local data), 377

datos miembro (data member), 435

decimal codificado en binario (binary-coded decimal), 59

declaración (declaration), 15

declaración de un dato (data declaration), 26

desarrollo (deployment), 712

desbordamiento aritmético (arithmetic overflow), 53

diagrama de flujo (flowchart), 115

diseño (design), 711

efectos laterales (side effects), 398

encapsulación (encapsulation), 435

entero (integer), 47

entrada de datos (data input), 16

enumerado (enumeration), 204

errores en tiempo de compilación (compilation error), 21

errores en tiempo de ejecución (execution error), 22

errores lógicos (logic errors), 22

espacio de nombres (namespace), 18

especificador de acceso (access specifier), 437

estado (state), 435

estado inválido (invalid state), 478

estilo camelcase, 29

estilo pascalcase, 364

estilo snake case, 29

estilo uppercamelcase, 29

estructura condicional (conditional structure), 116

estructura condicional doble (else conditional structure), 137

estructura repetitiva (iteration/loop), 210

estructura secuencial (sequential control flow structure), 116

evaluación en ciclo corto (short-circuit evaluation), 175

evaluación en ciclo largo (eager evaluation), 175  
 excepción (exception), 530, 702  
 exponente (exponent), 57  
 expresión (expression), 40  
 expresiones aritméticas (arithmetic expression), 65  
 fábrica (factory), 694  
 filtro (filter), 212  
 flujo de control (control flow), 20, 114  
 función (function), 16, 44  
 funciones globales (global functions), 431  
 funciones miembro (member functions), 435  
 genéricos (generics), 567  
 getline, 598  
 gigo: garbage input, garbage output, 32  
 google c++ style guide, 30  
 hardware, 2  
 identificador (identifier), 15  
 implementación (implementation), 711  
 implementación de un algoritmo (algorithm implementation), 9  
 indeterminación (undefined), 62  
 infinito (infinity), 62  
 ingeniería de software (software engineering), 715  
 instancia (instance), 434  
 interfaz (interface), 462  
 iso-10646, 87  
 iso/iec 8859-1, 86  
 iteración (iteration), 211  
 iteradores (iterators), 610  
 l-value, 41  
 lógico (boolean), 97  
 lectura anticipada, 222  
 lenguaje de programación (programming language), 7  
 lenguaje ensamblador (assembly language), 7  
 lenguajes de alto nivel (high level language), 7  
 leyes de de morgan (de morgan's laws), 133  
 lista de inicialización del constructor (constructor initialization list), 481  
 literal (literal), 33  
 literales de cadenas de caracteres (string literals), 33  
 literales de caracteres (character literals), 33  
 literales enteros (integer literals), 48  
 literales lógicos (boolean literals), 33  
 literales numéricos (numeric literals), 33  
 métodos (methods), 432, 435  
 módulo (module), 362



macro, 62  
mantenimiento (maintenance), 712  
mantisa (mantissa), 57  
marco de pila (stack frame), 387  
memoria dinámica (dynamic memory), 359  
modelo en cascada (waterfall model), 712  
modelos de desarrollo ágil (agile development models), 715  
modelos en espiral (spiral models), 714  
modelos incrementales (incremental models), 714  
modelos iterativos (iterative models), 713  
modularización (modularization), 430  
mutuamente excluyente (mutually exclusive), 146  
número de orden (code point), 85  
números mágicos (magic numbers), 229  
números pseudo-aleatorios (pseudorandom numbers), 491  
notación científica (scientific notation), 56  
notación infija (infix notation), 42  
notación prefija (prefix notation), 42  
objeto (object), 434  
objeto sin nombre (unnamed object), 664  
objetos (objects), 432  
operador (operator), 16, 43  
operador binario (binary operator), 42  
operador de asignación (assignment operator), 16  
operador de asignación por defecto (default assignment operator), 601  
operador de casting (casting operator), 79  
operador de resolución de ámbito (scope resolution operator), 542  
operador n-ario (n-ary operator), 42  
operador unario (unary operator), 42  
ordenación externa, 324  
ordenación interna, 324  
ordenación por inserción (insertion sort), 330  
ordenación por intercambio directo (burbuja) (bubble sort), 333  
ordenación por selección (selection sort), 326  
página de códigos (code page), 85  
parámetros (parameter), 44  
parámetros actuales (actual parameters), 365  
parámetros formales (formal parameters), 365  
paso de parámetro por valor (pass-by-value), 366

pila (stack), 387  
planificación (planning), 711  
plantillas (templates), 567  
precisión (precision), 60  
precondición (precondition), 426  
prevalencia de nombre (name hiding), 259  
principio de programación - ocultación de información (programming principle - information hiding), 405  
principio de programación - sencillez (programming principle - simplicity), 187  
principio de programación - una única vez (programming principle - once and only once), 111  
procedimiento (procedure), 393  
programa (program), 10  
programación (programming), 12  
programación declarativa (declarative programming), 431  
programación estructurada (structured programming), 431  
programación extrema (extreme programming), 715  
programación funcional (functional programming), 431  
programación modular (modular programming), 431  
programación orientada a objetos (object oriented programming), 432  
programación procedural (procedural programming), 431  
programador (programmer), 2  
pruebas (tests), 711  
pruebas de integración (integration tests), 712  
pruebas de unidad (unit testing), 712  
punteros (pointers), 676  
r-value, 41  
rango (range), 45  
real (real), 56  
redondeo (rounding), 58  
referencias (references), 676  
registro (record), 533  
reglas sintácticas (syntactic rules), 19  
salario bruto (gross income), 37  
salario neto (net income), 37  
salida de datos (data output), 17  
sentencia (sentence/statement), 15  
sentencia condicional (conditional statement), 117  
sentencia condicional doble (else conditional statement), 137  
sobrecarga de funciones (function overload), 396  
software, 2  
stack unwinding, 710  
struct, 533  
tamaño (size), 293

throw early catch late, **710**  
tipos de datos (data types), **15**  
tipos de datos primitivos (primitive  
data types), **26**  
transformación de tipo (casting), **67**

uml, **458**

unicode, **87**

uppercamelcase, **364**

usuario (user), **2**

validación (validation), **711**

valor (value), **16**

variables (variables), **33**

vector (array), **292**

verificación (verification), **711**