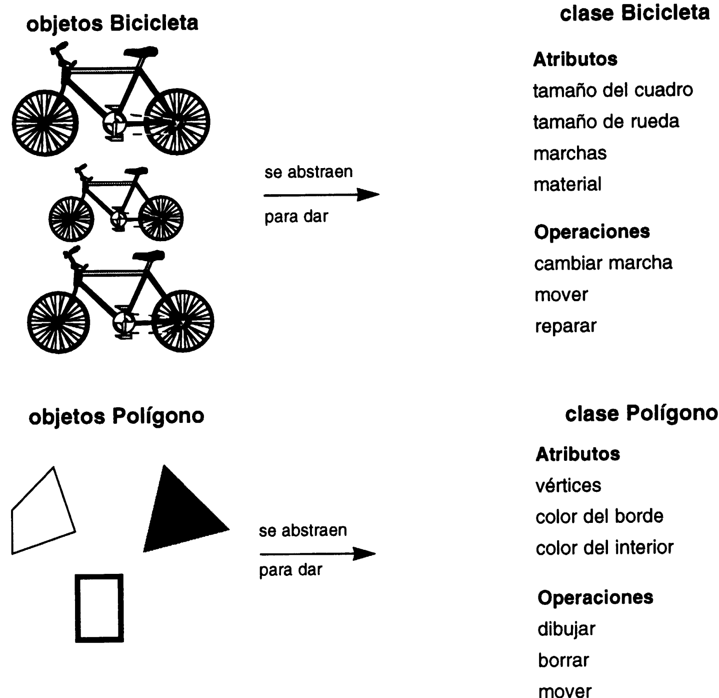
# Tema 4º
# Clases y objetos

- Implementación de clases y objetos
- Compartición estructural y copia de objetos
- Interacción entre clases y objetos
- Construcción de clases y objetos de calidad
- Resumen



# Implementación de clases y objetos

- Se implementará una clase Cola que se irá refinando en sucesivas versiones
- Todas las versiones se compilan y ejecutan correctamente pero tienen errores de diseño
- En este tema se repasan los conceptos del tema anterior desde un punto de vista práctico

## Ejemplo de manejo de la clase cola en C++ con constructores y destructores por defecto

```
#include <iostream.h>
const Maximo=100;

// Declaración de la clase cola
// Define como opera una cola de enteros

class cola {
  int c[Maximo];       // Por defecto todo es privado
  int ultimo, primero; // Indices del array
public:
  void inicializa(void);
  void meter(int);
  int sacar(void);
};

void cola::inicializa()
{
  primero = ultimo = 0;
}

void cola::meter(int valor)
{
  if(ultimo==Maximo) // Precondición
   {
    cout << "La cola está llena\n";
    return;
   }
  ultimo++;
  c[ultimo] = valor;
}

int cola::sacar(void)
{
  if(ultimo == primero) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  primero++;
  return c[primero];
}

int main(void)
{
  cola a, b;  // Se crean dos objetos de la clase cola

  a.inicializa();
  b.inicializa();
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << " ";
  cout << a.sacar() << " ";
  cout << b.sacar() << " ";
  cout << b.sacar() << "\n";
}
```

# Comentarios sobre el diseño de la versión 1 de la clase Cola

- El diseño del método inicializa()
  - Si se desea garantizar que inicializa() se ejecute cada vez que se crea un objeto esas operaciones deben estar dentro del constructor de la clase
- Se aconseja poner siempre constructores y destructores explícitos
  - Se puede colocar un simple mensaje
  - De esa forma sabemos siempre cuando se crean y se destruyen los objetos

## Ejemplo de constructores y destructores en C++

```
// Versión 2
#include <iostream.h>
const Maximo=100;

class cola {
  int c[Maximo];
  int ultimo, primero;
public:
  cola(void); //Constructor de la clase. No puede devolver valores
  ~cola(void); //Destructor de la clase.
  void meter(int);
  int sacar(void);
};

cola::cola(void)  //Sustituye al método inicializa de la versión 1
{
  primero = ultimo = 0;
  cout << "La cola ha quedado inicializada\n";
}
cola::~cola(void) //Se ejecuta automáticamente al finalizar el programa
{
  cout << "La cola ha quedado destruída\n";
}
void cola::meter(int valor)
{
  if(ultimo==Maximo) //Precondición
   {
    cout << "La cola está llena\n";
    return;
   }
  ultimo++;
  c[ultimo] = valor;
}
int cola::sacar(void)
{
  if(ultimo == primero) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  primero++;
  return c[primero];
}

int main(void)
{
  cola a, b;  // Se crean dos objetos de tipo cola
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << " ";
  cout << a.sacar() << " ";
  cout << b.sacar() << " ";
  cout << b.sacar() << "\n";
  return 0;
}
```

# Comentarios sobre el diseño de la versión 2 de la clase Cola

- Los objetos de clase Cola no están identificados
  - Si los objetos se almacenasen en disco no podríamos distinguir dos colas
- Se aconseja poner un atributo para identificar los objetos cola (versión 3)
- El atributo se pasa en el constructor
  - Un inconveniente es que no se garantiza que no se repitan identificaciones
  - Se puede hacer una versión con este atributo ***static*** y generado internamente por la propia clase

## Ejemplo de constructores con parámetros en C++

```
// Versión 3

#include <iostream.h>
const Maximo=100;
class cola {
  int c[Maximo];
  int ultimo, primero;
  int identificacion; //Permite distinguir las colas
public:
  cola(int);   //Constructor de la clase con un parámetro
  ~cola(void); //Destructor de la clase.
  void meter(int);
  int sacar(void);
};

cola::cola(int id)  //Constructor con parámetro. Versión 3.
{
  primero = ultimo = 0;
  identificacion=id;
  cout << "La cola "<<identificacion<<" ha quedado inicializada"<<endl;
}
cola::~cola(void) //Se ejecuta automáticamente al finalizar el programa
{
  cout << "La cola "<<identificacion<<" ha quedado destruída"<<endl;
}
void cola::meter(int valor)
{
  if(ultimo==Maximo) //Precondición
   {
    cout << "La cola está llena\n";
    return;
   }
  ultimo++;
  c[ultimo] = valor;
}
int cola::sacar(void)
{
  if(ultimo == primero) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  primero++;
  return c[primero];
}

int main(void)
{                 // Dos formas de paso de argumentos:
  cola a=cola(1); //Forma general
  cola b(2);      //Forma abreviada
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << " ";
  cout << a.sacar() << " ";
  cout << b.sacar() << " ";
  cout << b.sacar() << endl; //Observar la ejecución
}
```

# Comentarios sobre el diseño de la versión 3 de la clase Cola

- No se utiliza la modularidad de C++
  - Se separa en tres ficheros (versión 4)
    - `cola.hpp`. Donde se declaran las clases
    - `cola.cpp`. Donde se implementan los métodos
    - `prueba_cola_version_4.cpp`. Donde se realiza la prueba unitaria de la clase.
- Se implementa internamente con un array circular
- Se introduce un atributo `num_elementos` como auxiliar para manejar el array circular y para comprobar ciertas operaciones.
- Se introducen **postcondiciones**
  - La programación de postcondiciones se realiza con código redundante que verifica por partida doble la correción de las operaciones
- Se introducen métodos **selectores** e **iteradores**

```
// Capítulo Clases y Objetos: Ejercicio 4
// cola.hpp
//
// Ejemplo de métodos selector e iterador
//

// Versión 4
#ifndef COLA_HPP
#define COLA_HPP
//----------------------
#include <iostream.h>
//----------------------
const Maximo=3;

class cola {
  int c[Maximo];
  int ultimo, primero;
  int identificacion; //Permite distinguir las colas
  int num_elementos;
public:
  cola(int);   //Constructor de la clase con un parámetro
  ~cola(void); //Destructor de la clase.
  int estaVacia(void);//Método selector
  int estaLlena(void);//Método selector
  int longitud(void) {return num_elementos;};//Método selector
  void meter(int);  //Método modificador
  int sacar(void);  //Método modificador
  int ver_primero(void); //Método selector
  int ver_ultimo(void);   //Método selector
  int ver_identificacion(void){return identificacion;}; //Método selector
  int ver_posicion(int); //Método selector
  void mostrar_cola(void); //Método iterador
  void ver_array(void);//Método iterador utilizado solo para depurar
};
//-------------------
#endif
```

```
// Capítulo Clases y Objetos: Ejercicio 4
// cola.cpp
// Implementación de la clase cola
// con un array circular

// Versión 4

#include "cola.hpp"

cola::cola(int id)  //Constructor con parámetro
{
  primero = ultimo = 0;
  identificacion=id;
  num_elementos=0;
  cout << "El objeto cola "<<identificacion<<" ha sido creado"<<endl;
}

cola::~cola(void) // Se ejecuta automáticamente al finalizar el bloque
{
  cout << "El objeto cola "<<identificacion<<" ha sido destruido"<<endl;
}

int cola::estaVacia(void)// Método selector
{
  if(num_elementos==0) return 1;
   else return 0;
}

int cola::estaLlena(void)// Método selector
{
  if(num_elementos==Maximo) return 1;
   else return 0;
}

void cola::meter(int valor) //Método modificador
// Precondición: La cola no tiene que estar llena
// Modifica: Almacena valor en la cola
// Postcondición: La cola tiene un elemento más
{
  if(estaLlena()) //Precondición
   {
    cout <<"No se puede introducir el "<< valor;
    cout <<" la cola "<<ver_identificacion()<<" está llena"<<endl;
    return;
   }
  int num_elementos_antes=longitud();

  cout<<"Introduciendo "<<valor<<" en la cola "<<ver_identificacion()<<endl;



  c[ultimo] = valor;
  num_elementos++;

  ultimo++;
  // Si se alcanza el final del array se vuelve al principio
  if (ultimo==Maximo) ultimo=0;

  //Postcondición
  int num_elementos_despues=longitud();
  if (num_elementos_despues!=num_elementos_antes+1)
    {
     cout <<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento más"<<endl;
     return;
    }
}
```

```
int cola::sacar(void)
//Precondición: La cola no puede estar vacia
//Modifica: Devuelve y elimina de la cola el primer elemento
//Postcondición: La cola tiene un elemento menos
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola "<<ver_identificacion()<<" está vacía \n";
    return -1;
   }
  int num_antes=longitud();


  int valor=c[primero];
  num_elementos--;
  primero++;
  // Si se alcanza el final del array se vuelve al principio
  if (primero==Maximo) primero=0;

  cout<<"Sacando "<<valor<<" de la cola "<<ver_identificacion()<<endl;


  int num_despues=longitud();
  if (num_antes!=num_despues+1) //Postcondición
   {
     cout<<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento menos"<<endl;
     return -1;
    }
  return valor;
}

int cola::ver_primero(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }
  return c[primero];
}

int cola::ver_ultimo(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }

  if (ultimo>0) return c[ultimo-1];
   else return c[Maximo-1];
}
```

```
int cola::ver_posicion(int i)
{
  if(estaVacia()) //Primera precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }

  if (i>longitud()) //Segunda precondición
   {
    cout << "La cola solo tiene "<<longitud()<<endl;
    return -1;
   }
  if(primero+i-1<Maximo) return c[primero+i-1];
   else return c[primero+i-1-Maximo];
}

void cola::mostrar_cola(void)
{
  cout<<"La cola "<<ver_identificacion()<<" :";
  if (estaVacia())
   {
     cout<<"está vacía"<<endl;
     return;
    }
  for (int i=1; i<=longitud();i++)
     cout << ver_posicion(i)<<" ";
  cout <<endl;
}

void cola::ver_array(void)
 {
  for(int i=0;i<Maximo;i++)
  cout<<"c["<<i<<"]="<<c[i]<<endl;
  }
```

```
// Capítulo Clases y Objetos: Ejercicio 4
// prueba_cola_version_4.cpp
// Prueba unitaria de la clase cola
// Versión 4
#include "cola.hpp"
int main(void)
{
  cola a(1),b(2);
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  a.mostrar_cola();
  a.ver_array();
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  a.mostrar_cola();
  a.meter(500);
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  a.mostrar_cola();
  a.meter(1000);
  a.mostrar_cola();
  a.ver_array();
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  cout<<"Sacando elementos de la cola "<<b.ver_identificacion()<<endl;
  cout << b.sacar() << endl;
  cout << b.sacar() << endl;
  b.ver_array();
  b.mostrar_cola();
  b.meter(201);
  b.ver_array();
  b.meter(202);
  b.ver_array();
  b.meter(203);
  b.ver_array();
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  b.sacar();
  b.mostrar_cola();
  b.ver_array();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
  b.meter(204);
  b.mostrar_cola();
  b.ver_array();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
}
```

## Ejecución del Ejercicio 4 del capítulo Clases y Objetos

```
El objeto cola 1 ha sido creado
El objeto cola 2 ha sido creado
Introduciendo 10 en la cola 1
Introduciendo 19 en la cola 2
Introduciendo 20 en la cola 1
Introduciendo 1 en la cola 2
La cola 1 :10 20
c[0]=10
c[1]=20
c[2]=5234
La cola 1 tiene 2 elementos
El ultimo de la cola 1 es 20
El primero de la cola 1 es 10
La cola 1 :10 20
Introduciendo 500 en la cola 1
La cola 1 tiene 3 elementos
La cola 1 :10 20 500
No se puede introducir el 1000 la cola 1 está llena
La cola 1 :10 20 500
c[0]=10
c[1]=20
c[2]=500
El ultimo de la cola 1 es 500
El primero de la cola 1 es 10
Sacando elementos de la cola 2
Sacando 19 de la cola 2
19
Sacando 1 de la cola 2
1
c[0]=19
c[1]=1
c[2]=2640
La cola 2 :está vacía
Introduciendo 201 en la cola 2
c[0]=19
c[1]=1
c[2]=201
Introduciendo 202 en la cola 2
c[0]=202
c[1]=1
c[2]=201
Introduciendo 203 en la cola 2
c[0]=202
c[1]=203
c[2]=201
La cola 2 :201 202 203
Primero 201
Ultimo 203
Sacando 201 de la cola 2
La cola 2 :202 203
c[0]=202
c[1]=203
c[2]=201
Primero 202
Ultimo 203
La cola 2 tiene 2 elementos
Introduciendo 204 en la cola 2
La cola 2 :202 203 204
c[0]=202
c[1]=203
c[2]=204
Primero 202
Ultimo 204
La cola 2 tiene 3 elementos
El objeto cola 2 ha sido destruido
El objeto cola 1 ha sido destruido
```

# Comentarios sobre el diseño de la versión 4 de la clase Cola

- Siguiendo el principio de ocultación de la información se diseña una nueva clase Cola
  - La implementación interna es con punteros (parte privada)
  - Se mantienen los mismos métodos públicos
- Ahora el destructor no es un simple comentario
  - S e debe liberar memoria
  - Se vacían los objetos cola antes de destruirlos

```
// Capítulo Clases y Objetos: Ejercicio 5
// cola.hpp
//
// Clase cola implementada con punteros
//

// Versión 5


#ifndef COLA_HPP
#define COLA_HPP
//----------------------
#include <iostream.h>
//----------------------


struct nodo {
        int info;
        struct nodo *sig;
        };

typedef struct nodo NODO;

class cola {
  NODO *ultimo, *primero;
  int identificacion; //Permite distinguir las colas
  int num_elementos;
public:
  cola(int);   //Constructor de la clase con un parámetro
  ~cola(void); //Destructor de la clase.
  int estaVacia(void);//Método selector
  int longitud(void) {return num_elementos;};//Método selector
  void meter(int);  //Método modificador
  int sacar(void);  //Método modificador
  int ver_primero(void); //Método selector
  int ver_ultimo(void);   //Método selector
  int ver_identificacion(void){return identificacion;}; //Método selector
  int ver_posicion(int); //Método selector
  void mostrar_cola(void); //Método iterador
};
//-------------------
#endif
```

```
// Capítulo Clases y Objetos: Ejercicio 5
// cola.cpp
// Implementación de la clase cola
// con estructuras dinámicas de datos

// Versión 5

#include "cola.hpp"

cola::cola(int id)  //Constructor con parámetro
{
  primero = NULL;
  ultimo = NULL;
  identificacion=id;
  num_elementos=0;
  cout << "El objeto cola "<<identificacion<<" ha sido creado"<<endl;
}

cola::~cola(void) // Es necesario vaciarla
{
  int i;
// Vaciar la cola para liberar memoria
  while (!estaVacia()) sacar();
  cout << "El objeto cola "<<identificacion<<" ha sido destruido"<<endl;
}

int cola::estaVacia(void)// Método selector
{
  if(num_elementos==0) return 1;
   else return 0;
}

void cola::meter(int valor) //Método modificador
// Modifica: Almacena valor en la cola
// Postcondición: La cola tiene un elemento más
{
  int num_elementos_antes=longitud();
  cout<<"Introduciendo "<<valor<<" en la cola "<<ver_identificacion()<<endl;
  NODO *nuevo;
  nuevo=new NODO;  //Reserva dinámica de memoria para nuevo
  nuevo->info= valor;
  nuevo->sig=NULL;
  num_elementos++;

  if (ultimo==NULL)  //Caso de cola vacia
            {
     ultimo=nuevo;
     primero=nuevo;
    }
   else       //Caso en el que la cola ya tiene elementos
    {
     ultimo->sig=nuevo;
     ultimo=nuevo;
    }
  //Postcondición
  int num_elementos_despues=longitud();
  if (num_elementos_despues!=num_elementos_antes+1)
    {
     cout <<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento más"<<endl;
     return;
    }
}
```

```
int cola::sacar(void)
//Precondición: La cola no puede estar vacia
//Modifica: Devuelve y elimina de la cola el primer elemento
//Postcondición: La cola tiene un elemento menos
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola "<<ver_identificacion()<<" está vacía \n";
    return -1;
   }
  int num_antes=longitud();

  NODO *p;
  int valor=primero->info;
  p=primero;
  primero=primero->sig;
  delete p;    //Elimina p de la memoria dinámica
  if (primero==NULL) ultimo=NULL;
  num_elementos--;

  cout<<"Sacando "<<valor<<" de la cola "<<ver_identificacion()<<endl;

  int num_despues=longitud();
  if (num_antes!=num_despues+1) //Postcondición
   {
     cout<<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento menos"<<endl;
     return -1;
    }
  return valor;
}

int cola::ver_primero(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }
  return primero->info;
}

int cola::ver_ultimo(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  return ultimo->info;
}
```

```
int cola::ver_posicion(int i)
{
  if(estaVacia()) //Primera precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }

  if (i>longitud()) //Segunda precondición
   {
    cout << "La cola solo tiene "<<longitud()<<endl;
    return -1;
   }

  NODO *p;
  p=primero;
  for (int j=1;j<i;j++) p=p->sig;
  return p->info;
}

void cola::mostrar_cola(void)
{
  cout<<"La cola "<<ver_identificacion()<<" :";
  if (estaVacia())
   {
     cout<<"está vacía"<<endl;
     return;
    }
  for (int i=1; i<=longitud();i++)
     cout << ver_posicion(i)<<" ";
  cout <<endl;
}
```

```
// Capítulo Clases y Objetos: Ejercicio 5
// c_o_e5.cpp
// Prueba unitaria de la clase cola
// Versión 5

#include "cola.hpp"
int main(void)
{
  cola a(1),b(2);

  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  a.mostrar_cola();
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  a.mostrar_cola();
  a.meter(500);
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  a.mostrar_cola();
  a.meter(1000);
  a.mostrar_cola();
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  cout<<"Sacando elementos de la cola "<<b.ver_identificacion()<<endl;
  cout << b.sacar() << endl;
  cout << b.sacar() << endl;
  b.mostrar_cola();
  b.meter(201);
  b.meter(202);
  b.meter(203);
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  b.sacar();
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
  b.meter(204);
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
}
```

### Ejecución del ejercicio 5 del capítulo Clases y Objetos

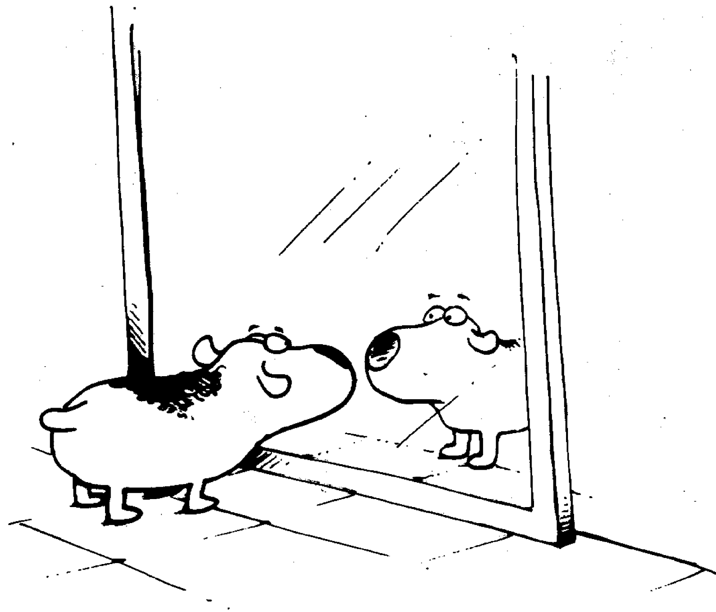
```
El objeto cola 1 ha sido creado
El objeto cola 2 ha sido creado
Introduciendo 10 en la cola 1
Introduciendo 19 en la cola 2
Introduciendo 20 en la cola 1
Introduciendo 1 en la cola 2
La cola 1 :10 20
La cola 1 tiene 2 elementos
El ultimo de la cola 1 es 20
El primero de la cola 1 es 10
La cola 1 :10 20
Introduciendo 500 en la cola 1
La cola 1 tiene 3 elementos
La cola 1 :10 20 500
Introduciendo 1000 en la cola 1
La cola 1 :10 20 500 1000
El ultimo de la cola 1 es 1000
El primero de la cola 1 es 10
Sacando elementos de la cola 2
Sacando 19 de la cola 2
19
Sacando 1 de la cola 2
1
La cola 2 :está vacía
Introduciendo 201 en la cola 2
Introduciendo 202 en la cola 2
Introduciendo 203 en la cola 2
La cola 2 :201 202 203
Primero 201
Ultimo 203
Sacando 201 de la cola 2
La cola 2 :202 203
Primero 202
Ultimo 203
La cola 2 tiene 2 elementos
Introduciendo 204 en la cola 2
La cola 2 :202 203 204
Primero 202
Ultimo 204
La cola 2 tiene 3 elementos
El objeto cola 2 ha sido destruido
El objeto cola 1 ha sido destruido
```

# Compartición estructural

- *Se produce cuando la identidad de un objeto recibe un alias a través de un segundo nombre*
- *El mismo objeto tiene dos identificadores*
- *En lenguajes como C++ se permite el uso de alias o referencias*
- *No es aconsejable el uso de alias o de referencias que no sean parámetros de funciones*

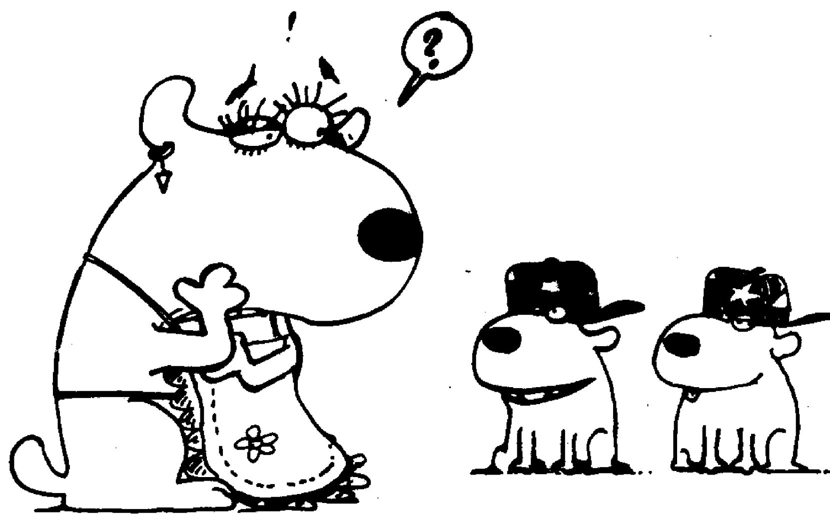
## Ejemplo de referencias independientes en C++

```
class Persona
 {…};
Persona francisco;
Persona &paco=francisco; //paco y francisco
                         //son el mismo objeto
```



# Copia de objetos

- *Se pretende obtener dos objetos con identificadores distintos pero con estados iguales*
- *En C++ no debe utilizarse el operador asignación (=) para crear copias de objetos, pues para objetos cuyo estado involucra a punteros a otros objetos, sólo se copia el puntero pero no a lo que apunta el puntero.*
- *Las soluciones son:*
  - Añadir a la clase constructores de copia
  - Sobrecargar el operador asignación en la clase para que realice la copia correctamente
- *Para determinar si dos objetos son iguales se deben sobrecargar los operadores de comparación == y !=*



## Constructores de copia por defecto en C++

- Un constructor de copia es un constructor de la clase que se utiliza para copiar objetos de dicha clase
- C++ incorpora genera automáticamente un constructor de copia defecto (también denominado *constructor de copia por omisión*)
- Sólo se generará un constructor de copia por defecto si no se ha declarado un constructor de copia
- El constructor de copia por defecto se utiliza automáticamente en los siguientes casos:
    - En una asignación entre objetos de la misma clase
    - Cuando se pasa por valor un objeto de la clase
    - Cuando una función devuelve un objeto de la clase

```
// Capítulo Clases y Objetos: Ejercicio 6
// c_o_e6.cpp
// Uso del constructor de copia por defecto
// Versión 6
#include "cola.hpp"
int main(void)
{
  cola a(1),b(2);

  a.meter(10);
  a.meter(20);
  a.mostrar_cola();
  b=a;// la asignación utiliza el constructor de copia por defecto
  b.mostrar_cola();
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
  cout <<"El ultimo de la cola "<<b.ver_identificacion();
  cout <<" es "<<b.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<b.ver_identificacion();
  cout <<" es "<<b.ver_primero()<<endl;
}
```

**Ejecución**

```
El objeto cola 1 ha sido creado
El objeto cola 2 ha sido creado
Introduciendo 10 en la cola 1
Introduciendo 20 en la cola 1
La cola 1 :10 20
La cola 1 :10 20
La cola 1 tiene 2 elementos
El ultimo de la cola 1 es 20
El primero de la cola 1 es 10
El objeto cola 1 ha sido destruido
El objeto cola 1 ha sido destruido
```

# Constructores de copia en C++

• Un constructor de copia es un constructor de la clase que se utiliza para copiar objetos de dicha clase

• Los constructores de copia son de la forma: `C(C&)` o `C(const C&)` donde C es el nombre de la clase.

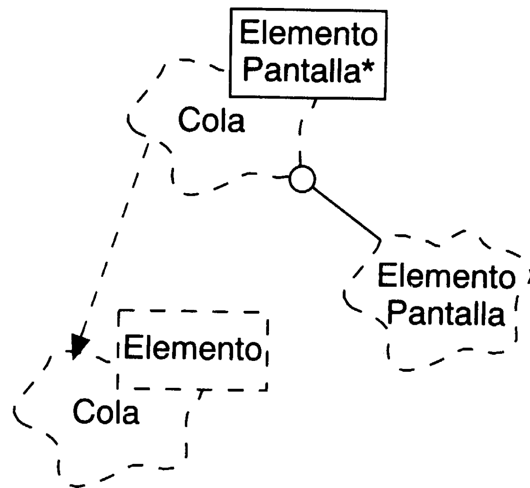
```
// Capítulo Clases y Objetos: Ejercicio 7
// cola.hpp
//
// Ejemplo con constructor de copia y sobrecarga de operadores
//

// Versión 7
#ifndef COLA_HPP
#define COLA_HPP
//----------------------
#include <iostream.h>
//----------------------

struct nodo {
        void *info; //Cola genérica de punteros a void
        struct nodo *sig;
        };

typedef struct nodo NODO;

class cola {
  NODO *ultimo, *primero;
  int identificacion; //Permite distinguir las colas
  int num_elementos;
public:
  cola(int=0);   //Constructor de la clase con un parámetro por defecto
  cola(const cola&); //Constructor de copia
  cola& operator = (const cola&); // Sobrecarga de la asignación
  int operator == (const cola&); // Sobrecarga del operador comparación ==
  int operator != (const cola&); // Sobrecarga del operador comparación !=
  ~cola(void); //Destructor de la clase.
  int estaVacia(void);//Método selector
  int longitud(void) {return num_elementos;};//Método selector
  void meter(const void*);  //Método modificador
  const void* sacar(void);  //Método modificador
  const void* ver_primero(void); //Método selector
  const void* ver_ultimo(void);   //Método selector
  int ver_identificacion(void){return identificacion;}; //Método selector
  const void* ver_posicion(int); //Método selector
};
//-------------------
#endif
```

# Genericidad

- La genericidad es soportada por lenguajes OO como C++ y Eiffel mediante el uso de clases parametrizadas (“plantillas” o “templates”)
- Las clases parametrizadas permiten escribir una plantilla que se pueden aplicar a varios casos que difieren solamente en los tipos de los parámetros
- Una clase parametrizada no puede tener instancias a menos que antes se la instancie indicando los parámetros
- Usando el ejemplo en C++ para definir dos objetos cola

```
Cola<int> colaEnteros;
Cola<ElementoPantalla*>colaElementos;
```

- Las clases parametrizadas son seguras respecto al conrol de tipos que se determina en tiempo de compilación



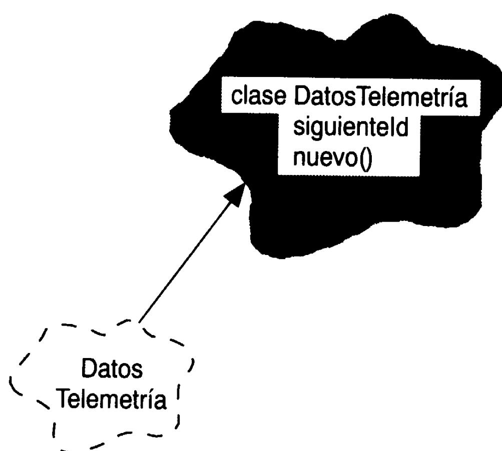

Notación de Booch

Ejemplo en C++

```
template<class Elemento>
class Cola{
public:
            Cola();
            Cola(const Cola<Elemento>&);
            virtual ~Cola();
virtual Cola<Elemento>& operator=(const Cola<Elemento>&);
virtual int operator==(const Cola<Elemento>&)const;
int operator !=(const Cola<Elemento>&) const;
virtual void borrar();
virtual void anadir(const Elemento&);
. . .
}
```

# Metaclases

- *Una metaclase es una clase cuyas instancias son clases*
- Los lenguajes Smalltalk y CLOS soportan metaclases
- Los lenguajes C++, Object Pascal, Ada, Eiffel no lo soportan
- El problema es que se deben crear clases en tiempo de ejecución
- La notación de Booch da soporte a las metaclases con una nube gris y una flecha gris.




Metaclases.

# Interacción entre clases y objetos

- **Relaciones entre clases y objetos**
  - Las clases y objetos son conceptos separados pero muy relacionados entre sí
  - Todo objeto es instancia de alguna clase
  - Toda clase tiene cero o más instancias
  - ***Las clases son estáticas***. Su existencia y semántica están fijadas antes de la ejecución del programa. La única excepción son los lenguajes que usan metaclases
  - ***Los objetos son dinámicos***. Se crean y se destruyen en tiempo de ejecución, un caso particular son los objetos persistentes
- **El papel de clases y objetos en análisis y diseño**. Durante el análisis y las primeras etapas del diseño el desarrollador tiene dos tareas principales:
  - *Identificar las clases y objetos que forman el vocabulario del dominio del problema (**abstracciones clave**)*
  - *Idear las estructuras por las que conjuntos de objetos trabajan juntos para lograr los comportamientos que satisfacen los requisitos del problema (**mecanismos**)*

# Construcción de clases y objetos de calidad

- Medida de la calidad de una abstracción
  - Acoplamiento *es la medida de la fuerza de la asociación establecida por una conexión entre un módulo y otro*
    - Acoplamiento entre módulos débil reduce la complejidad
    - Acoplamiento entre clases y objetos versus herencia (acoplamiento fuerte)
  - Cohesión *mide el grado de conectividad entre los elementos de un sólo módulo y en DOO la de los elementos de una clase*
    - La peor: **cohesión por coincidencia** *(las abstracciones de un mismo módulo o clase no tienen ninguna relación)*
    - La mejor: **cohesión funcional** *(las abstracciones trabajan conjuntamente para proporcionar un comportamiento determinado)*
  - Suficiencia: *la clase o módulo captura suficientes características de la abstracción como para permitir una interacción significativa y eficiente*
  - Completud (estado completo o plenitud): *El interfaz de la clase o módulo captura todas las características significativas de la abstracción*
  - Operaciones primitivas: *Sólo se implementan en la clase o el módulo aquellas operaciones cuya implementación eficiente sólo se produce si se tiene acceso a la representación interna de la abstracción*

# Construcción de clases y objetos de calidad (II)

- Selección de operaciones
  - Semántica funcional: *Seleccionar el interfaz de una clase es complejo. Un buen diseñador debe llegar a un equilibrio entre subdividir demasiado en subclases y dejar clases o módulos demasiado grandes*
  - Semántica espacial: *Especifica la cantidad de espacio de almacenamiento de cada operación*
  - Semántica temporal: *Especifica la semántica de la concurrencia en términos de sincronización.*
- Elección de relaciones
  - Colaboraciones
    - Los métodos de una clase tan sólo deben depender de la propia clase
    - Un método sólo debe enviar mensajes a objetos de un número limitado de clases
    - La jerarquía de herencias puede ser en forma de bosque (debilmente acopladas) y árboles (acoplamiento fuerte). La jerarquía de herencias elegida depende del tipo de problema
  - Mecanismos y visibilidad
    - Las relaciones entre objetos plantean el diseño de mecanismos que reflejan como los objetos interactúan
    - Es útil definir como un objeto es visible para otros

# Construcción de clases y objetos de calidad (III)

- Elección de implementaciones
  - Representación
    - *La representación interna de una clase es un secreto encapsulado de la abstracción*
    - *La elección de la representación es difícil y no unívoca*
    - *La modificación de la representación interna no debe violar ninguno de los contratos con los clientes de la clase*
  - Empaquetamiento
    - *Elegir las clases y objetos que se empaquetan en cada módulo*
    - *Los requisitos de ocultación y visibilidad entre módulos suelen guiar las decisiones de diseño*
    - *Generalmente los módulos tienen cohesión funcional y están débilmente acoplados entre sí*
    - *Hay factores no técnicos que influyen en la definición de módulos: reutilización, seguridad y documentación.*

# Resumen

- Un objeto tiene estado, comportamiento e identidad
- La estructura y comportamiento de objetos similares están definidos en su clase común
- El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades
- El comportamiento es la forma que un objeto actúa y reacciona en términos de sus cambios de estado y paso de mensajes
- La identidad es la propiedad de un objeto que lo distingue de todos los demás objetos
- Los dos tipos de jerarquías de objetos son relaciones de asociación y agregación
- Una clase es un conjunto de objetos que comparten una estructura y comportamiento comunes
- Los seis tipos de jerarquías de clase son las relaciones de asociación, herencia, agregación, “uso”, instanciación, clases parametrizadas y relaciones de metaclase
- Las abstracciones clave son las clases y objetos que forman el vocabulario del dominio del problema
- Un mecanismo es una estructura por la que un conjunto de objetos trabajan juntos para ofrecer un comportamiento que satisfaga algún requisito del problema
- La calidad de una abstracción puede medirse por su acoplamiento, cohesión, suficiencia, completud (estado completo) y las operaciones primitivas

# Autoevaluación (I)

- Un iterador es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un selector es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un modificador es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un constructor es **A)** Una operación que libera el estado de un objeto y/o destruye el propio objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un destructor es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- El tiempo de vida de un objeto **A)** Se extiende desde que lo crea un constructor hasta que se destruye por un destructor **B)** En lenguajes con recolección de basura se extiende desde que se crea hasta que todas las referencias a el objeto se han perdido **C)** Si el objeto es persistente el tiempo de vida transciende a la vida del programa que lo crea **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Relaciones entre clases y objetos **A)** Las clases en C++ son estáticas **B)** Los objetos en C++ son dinámicos **C)** En C++ todo objeto es instancia de alguna clase **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un constructor de una clase en C++ **A)** Siempre existe implícita o explícitamente **B)** Es una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un destructor **A)** Es una operación que no existe en los lenguajes con recolección de basura **B)** No existe en Java **C)** Siempre existe en C++ **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta

# Autoevaluación (II)

- Sea el siguiente fragmento de código en C++ que esta en un módulo denominado `avion.h`

```
class Ala {...}; // Los ... indican que se ha implementado la clase
class Motor {...};
class TrenDeAterrizaje{...};
class Cabina{...};
class Avion {
     Ala *a1, *a2;
     Motor *m1, *m2;
     TrenDeAterrizaje *t;
     Cabina *c;
     ...}
```

Se puede decir **A)** Que la clase Avion hereda de las anteriores **B)** Que `Ala` es un método virtual de la clase Avion **C)** Que la clase `Avion` es una agregación **D)** Todas las respuestas anteriores son correctas **E)** Todas las respuestas son falsas.

- Utilizando la clase Avion anterior se escribe el siguiente fragmento de código:

```
#include  "avion.h"
main ()
 {
  Avion A310;
  ...
 }
```

Se puede afirmar **A)** La instrucción `Avion A310;` es una llamada a un método constructor **B)** A310 es un objeto **C)** A310 se crea en tiempo de ejecución **D)** Todas las respuestas anteriores son correctas **E)** Todas las respuestas anteriores son falsas.

- Siguiendo con el código del apartado anterior se puede afirmar **A)** No se ejecuta el destructor **B)** Si se ejecuta el destructor **C)** Para que se ejecute el destructor debe escribirse obligatoriamente en C++ **D)** El objeto A310 es inmortal y no se destruye nunca **E)** Todas las respuestas anteriores son falsas.

# Referencias


- [Booch 94] G.Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings (1994). Versión castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- [Cueva 93] Cueva Lovelle, J.M. García Fuente Mª P., López Pérez B., Luengo Díez Mª C., Alonso Requejo M. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Distribuido por Ciencia-3 (1993).
- [Joyanes 96] L. Joyanes Aguilar. *Programación orientada a objetos. Conceptos, modelado, diseño y codificación en C++*. McGraw-Hill (1996).
- [Meyer 97] B. Meyer *Object-oriented software construction.* Second Edition. Prentice-Hall (1997). Versión castellana: *Construcción de software orientado a objetos.* Prentice-Hall (1998).
- [Rational 97] UML y herramienta Rational/Rose en *www.rational.com*
- [Rumbaught 91] Rumbaught J., Blaha M., Premerlani W., Wddy F., Lorensen W. *Object-oriented modeling and design.* Prentice-Hall (1991). Versión castellana: *Modelado y diseño orientado a objetos. Metodología OMT*. Prentice-Hall (1996)