

Tema 2: Programación Orientada a Objetos con C++

Tecnología de la Programación de Videojuegos 1
Grado en Desarrollo de Videojuegos

Miguel Gómez-Zamalloa Gil

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Programación Orientada a Objetos

- ♦ Los programas crecen ... y mucho ...
- ♦ La POO ofrece una solución al problema del diseño, división de tareas y organización de los programas
- ♦ En los años 90 se convirtió en el paradigma más usado
 - ❖ Ver entrevista Steve Jobs: <https://youtu.be/HNMAXCfP6K4> (minuto 59)
- ♦ Ideas fundamentales de la POO:
 - ▶ Un objeto incluye datos (su estado) y comportamiento (lo que sabe hacer)
 - ▶ Cualquier actividad no trivial se realiza por interacción de una comunidad de objetos que cooperan
 - ▶ Encapsulación: Separación total entre interfaz e implementación
- ♦ La POO permite fundamentalmente:
 - ▶ División real de tareas
 - ▶ Claridad y modularidad
 - ▶ Reutilización: Tanto las propias piezas como por herencia de ellas

Principales Diferencias con C#

♦ Manejo de memoria:

- ▶ Se pueden manejar punteros a objetos (como en C#) pero también objetos in situ. Esto tiene varias repercusiones:
 - ❖ Constructoras por defecto y por copia
 - ❖ Operador de asignación
- ▶ Destructoras

♦ Distribución del código en ficheros .h y .cpp

♦ Operadores

♦ Genericidad mediante plantillas

♦ Más diferencias:

- ▶ Las clases no tienen método main
- ▶ En C++ no hay interfaces
- ▶ Herencia múltiple
- ▶ Etc.

Interfaz vs. Implementación

Interfaz

- ♦ Todo lo que el usuario de la clase necesita saber
 - ❖ Tipos de datos
 - ❖ Declaraciones de métodos y/o funciones
 - ❖ Declaración de constantes

Implementación

- ♦ Código de los métodos y/o funciones de la interfaz

- ♦ En C++ se separa la interfaz e implementación en dos archivos:
 - ▶ Archivo de cabecera o interfaz (**.h**): Tipos de datos, declaración de la clase (incluyendo declaración de todos los métodos, incluso los privados)
 - ▶ Archivo de implementación (**.cpp**): Código de las funciones y/o métodos.
- ♦ Distribución del código en ficheros **.h** y **.cpp**
- ♦ El usuario de la clase solo hará **#include** del fichero **.h**

Ejemplo: Vector2D

Vector2D.h

Normalmente lo escriben los IDEs.
Visual escribe **#pragma once**

```
#ifndef VECTOR2D_H_
#define VECTOR2D_H_
#include <iostream>

class Vector2D {
private:
    double x;
    double y;
public:
    Vector2D();
    Vector2D(double x, double y);
    double getX() const;
    double getY() const;
    void normalize();
    Vector2D operator+(const Vector2D& v) const;
    Vector2D operator*(double d) const;
    double operator*(const Vector2D& d) const;
    friend std::ostream& operator<<(std::ostream& os, const Vector2D& v);
};

#endif // También lo escriben los IDEs (va con el #ifndef de arriba)
```

Ejemplo: Vector2D

Vector2D.cpp

```
#include "Vector2D.h"
#include <math.h>

Vector2D::Vector2D() : x(), y() {}

Vector2D::Vector2D(double x, double y) : x(x), y(y) {}

double Vector2D::getX() const {
    return x;
}

double Vector2D::getY() const {
    return y;
}

void Vector2D::normalize() {
    double mag = sqrt(pow(x, 2) + pow(y, 2));
    if (mag > 0.0) {
        x = x / mag;
        y = y / mag;
    }
}
```

Ejemplo: Vector2D

Vector2D.cpp

```
Vector2D Vector2D::operator+(const Vector2D& v) const {  
    Vector2D r;  
    r.x = this->x + v.x; // El this no es necesario. Se pone para ilustrar su uso  
    r.y = this->y + v.y; // Al ser un puntero se debe usar con ->  
    return r;  
}
```

```
Vector2D Vector2D::operator*(double d) const {  
    Vector2D r;  
    r.x = x * d;  
    r.y = y * d;  
    return r;  
}
```

```
double Vector2D::operator*(const Vector2D& d) const {  
    return d.x * x + d.y * y;  
}
```

```
std::ostream& operator<<(std::ostream& os, const Vector2D &v) {  
    os << "(" << v.x << "," << v.y << ")";  
    return os;  
}
```

Ejemplo: Vector2D

main.cpp

```
#include "Vector2D.h"

int main(){
    Vector2D a(1,1);
    Vector2D b = Vector2D(2,2);
    a = a*2;
    Vector2D c = a+b;
    c.normalize();
    cout << "El prod. escalar de "
         << a << " y " << b << " es "
         << a*b << endl;
}
```


Ejemplo: VectorOfDoubles

```
#ifndef VECTOROFDOUBLES_H // Para evitar inclusiones múltiples. Lo ponen los IDEs
#define VECTOROFDOUBLES_H // Todo el código debe ir entre este punto y el #endif
```

```
using uint = unsigned int; // Alias por comodidad
```

```
class VectorOfDoubles {
```

```
private:
```

```
    static const uint DEFAULT_CAPACITY = 5; // Capacidad inicial por defecto
```

```
    uint capacity; // Capacidad actual del array dinámico
```

```
    uint numElems = 0; // Contador del número de elementos
```

```
    double* elems; // Array dinámico de elementos double
```

```
public:
```

```
    VectorOfDoubles() : capacity(DEFAULT_CAPACITY), elems(new double[capacity]) {}
```

```
    ~VectorOfDoubles() { delete[] elems; numElems = 0; elems = nullptr; }
```

```
    uint size() const { return numElems; }
```

```
    double operator[](int i) const { return elems[i]; }
```

```
    double& operator[](int i) { return elems[i]; }
```

```
    bool empty() const { return numElems == 0; }
```

```
    ...
```

Ver slide 14

VectorOfDoubles.h

Ejemplo: VectorOfDoubles

VectorOfDoubles.h

...

void push_back(double e); // Pone nuevo elemento al final

void pop_back(); // Quita último elemento

bool insert(double e, uint i); // Inserta e en posición iésima (desplazando)

bool erase(uint i); // Borra el elemento de la pos. iésima (desplazando)

private:

void reallocate();

void shiftRightFrom(uint i);

void shiftLeftFrom(uint i);

};

#endif // También lo escriben los IDEs (va con el #ifndef del principio)

Ejemplo: VectorOfDoubles

```
#include "VectorOfDoubles.h"
```

VectorOfDoubles.cpp

```
void VectorOfDoubles::push_back(double e){  
    if (numElems == capacity) reallocate();  
    elems[numElems] = e;  
    ++numElems;  
}
```

```
void VectorOfDoubles::pop_back(){  
    if (numElems > 0) --numElems;  
}
```

```
bool VectorOfDoubles::insert(double e, uint i){  
    if (i > numElems) return false;  
    else {  
        if (numElems == capacity) reallocate();  
        shiftRightFrom(i);  
        elems[i] = e;  
        ++numElems;  
        return true;  
    }  
}  
...  
...
```

Ejemplo: VectorOfDoubles

VectorOfDoubles.cpp

```
...  
bool VectorOfDoubles::erase(uint i){  
    if (i >= numElems) return false;  
    else {  
        shiftLeftFrom(i);  
        --numElems;  
        return true;  
    }  
}  
  
// Métodos privados  
  
void VectorOfDoubles::shiftRightFrom(uint i){  
    for (uint j = numElems; j > i; j--)  
        elems[j] = elems[j-1];  
}  
  
void VectorOfDoubles::shiftLeftFrom(uint i){  
    for (; i < numElems-1; i++)  
        elems[i] = elems[i+1];  
}  
...
```

Ejemplo: VectorOfDoubles

VectorOfDoubles.cpp

```
...  
void VectorOfDoubles::reallocate(){  
    capacity = capacity*2;  
    double* newElems = new double[capacity];  
    for (uint i = 0; i < size(); i++)  
        newElems[i] = elems[i];  
    delete[] elems;  
    elems = newElems;  
}
```


Destructor

- ✦ Toda clase tiene un método especial denominado **destructor** cuyo rol es liberar la memoria dinámica creada por el objeto
- ✦ Si no se proporciona el compilador genera una destructora vacía
- ✦ La destructora se invoca automáticamente cuando:
 - ▶ Un objeto (no puntero) sale de ámbito
 - ▶ Cuando se destruye con delete a su puntero un objeto dinámico
 - ▶ Salvo excepciones, el programador nunca debe invocarla explícitamente
- ✦ Tras ejecutar la destructora se invocan automáticamente las destructoras de los atributos de tipo objeto (no punteros)
 - ❖ Esto es un caso particular de salida de ámbito

✦ Ejemplo:

```
~VectorOfDoubles() {  
    delete[] elems;  
    numElems = 0;  
    elems = nullptr;  
}
```

Destructora

```
class A {  
    int i; A a5;   
    A(int i) : i(i) {  
        cout << "ctor a" << i << '\n';  
    }  
    ~A(){  
        cout << "dctor a" << i << '\n';  
    }  
};  
  
A a0(0);  
  
int main(){  
    A a1(1);  
    A* p;  
    { // nuevo ámbito  
        A a2(2);  
        p = new A(3);  
    } // a2 sale de ámbito  
    delete p; // llama al destructor de a3  
}
```

Qué pasa si añadimos un atributo de tipo A?

Salida del programa:

```
ctor a0  
ctor a1  
ctor a2  
ctor a3  
dctor a2  
dctor a3  
dctor a1  
dctor a0
```


Punteros a Instancias vs. Instancias

- ✦ En C++ se pueden tener punteros a instancias (como en C# o Java) pero también se pueden manejar instancias in situ

```
int main(){
    VectorOfDoubles v; // se ejecuta la constructora
    VectorOfDoubles* pv = &v; // puntero a v
    pv = new VectorOfDoubles; // se ejecuta la constructora para *pv
    v.push_back(1);
    pv->push_back(2); // operador -> para abreviar (*pv).push_back(2)
    cout << v.size();
    cout << pv->size(); // (*pv).size()
    delete pv; // se ejecuta la destructora
}
```

- ✦ Esto tiene implicaciones importantes:
 - ▶ Mayor control ➡ potencialmente más eficiencia
 - ▶ Mayor peligro ➡ se manifiesta en clases con manejo de memoria dinámica

Constructor por Copia y Asignación

- ◆ Consideremos estos bloques de código, aparentemente inofensivos:

```
{  
  VectorOfDoubles v1;  
  VectorOfDoubles v2 = v1;  
}
```

```
{  
  VectorOfDoubles v1;  
  VectorOfDoubles v2;  
  v2 = v1;  
}
```

```
VectorOfDoubles v1;  
VectorOfDoubles v2 = v1;  
for (int i = 0; i < 6; ++i)  
  v1.push_back(i);
```

- ◆ Todos dan lugar a **errores de ejecución** (en modo Debug), y aún peor, pueden pasar inadvertidos y producir **comportamientos impredecibles**
- ◆ **Constructor por copia:**
 - ▶ Se invoca automáticamente en inicialización por copia, paso de parámetros por copia y en return's
 - ▶ Si no se define, el compilador genera uno por defecto (copia superficial)
- ◆ **Operador de asignación (operator=):**
 - ▶ Se invoca cuando se asigna una instancia con otra
 - ▶ Si no se define, el compilador genera uno por defecto (copia superficial)

Constructor por Copia y Asignación

- ✦ El problema viene por la compartición parcial de memoria
- ✦ En C# o Java también puede ocurrir, aunque es mucho más difícil
 - ▶ Por ejemplo, si se hace una copia (con clone) superficial
- ✦ Posibles soluciones:
 - ▶ Implementar el constructor por copia y el operador de asignación de manera que hagan una copia profunda
 - ❖ Ojo: El operador de asignación debe borrar la instancia antigua
 - ❖ Las "move semantics" dan una solución eficiente para estos casos
 - ▶ Programar estilo C# o Java, usando siempre punteros a instancias
 - ❖ Para evitar su uso, se pueden suprimir (mediante "`= delete`"), poner privados o hacer que lancen excepción.
- ✦ El problema se estudia también en EDA

Constructor por Copia y Asignación

```
class VectorOfDoubles {  
public:  
    VectorOfDoubles() : capacity(DEFAULT_CAPACITY), elems(new double[capacity]) {}  
  
    VectorOfDoubles(const VectorOfDoubles& other) { copy(other); };  
    //VectorOfDoubles(const VectorOfDoubles& other) = delete;  
  
    ~VectorOfDoubles() { free(); numElems = 0; elems = nullptr; }  
  
    VectorOfDoubles& operator=(const VectorOfDoubles& other){  
        if (this != &other) {  
            free();  
            copy(other);  
        }  
        return *this;  
    }  
    //VectorOfDoubles& operator=(const VectorOfDoubles& other) = delete;  
    ...  
};
```

Constructor por Copia y Asignación

```
class VectorOfDoubles {  
    ...  
  
private:  
    void free() { //Iría al .cpp  
        delete[] elems;  
    }  
  
    void copy(const VectorOfDoubles& other){ //Iría al .cpp  
        capacity = other.capacity;  
        numElems = other.numElems;  
        elems = new T[capacity];  
        for (unsigned int i = 0; i < numElems; ++i)  
            elems[i] = other.elems[i];  
    }  
}
```

Ejemplo: Vector Genérico

```
#ifndef VECTOR_H_ // Para evitar inclusiones múltiples. Lo suelen poner los IDEs
#define VECTOR_H_ // Todo el código debe ir entre este punto y el #endif (ver abajo)
```

```
template <class T> // o template <typename T>
```

Vector.h

```
class Vector {
```

```
private:
```

```
    static const uint DEFAULT_CAPACITY = 5; // Capacidad inicial por defecto
```

```
    uint capacity; // Capacidad actual del array dinámico
```

```
    uint numElems = 0; // Contador del número de elementos
```

```
    T* elems; // Array dinámico de elementos de tipo T
```

```
public:
```

```
    Vector() : capacity(DEFAULT_CAPACITY), elems(new T[capacity]) {}
```

```
    ~Vector() {delete[] elems; numElems = 0; elems = nullptr;}
```

```
    uint size() const {return numElems;}
```

```
    const T& operator[](int i) const {return elems[i];}
```

```
    T& operator[](int i){return elems[i];}
```

```
    void push_back(const T& e); // Pone nuevo elemento al final
```

```
    bool insert(const T& e, uint i); // Inserta e en posición iésima (desplazando)
```

```
    ...
```

Ejemplo: Vector Genérico

Vector.h

```
template <class T>
class Vector {
private:
    T* elems;
    ...
};
```

// En las plantillas el código de los métodos va también en el .h

```
template<class T>
void Vector<T>::push_back(const T& e){
    if (numElems == capacity) reallocate();
    elems[numElems] = e;
    ++numElems;
}
...
```

◆ Instanciación:

La destructora de v2 borra el array dinámico pero no el Vector2D

```
Vector<Vector2D> v; // Instanciación. Igual que en C#
v.push_back(Vector2D(1,1)); // Se usa igual
v[0].normalize();
```

```
Vector<Vector2D*> v2;
v2.push_back(new Vector2D(2,2));
delete v2[0];
```


Atributos y Métodos de Clase (static)

Atributos estáticos:

- ♦ No forman parte del estado de los objetos
- ♦ Los pueden utilizar todos los objetos: una única copia para todos los objetos de la clase
- ♦ Es obligatorio inicializarlas
- ♦ Útil para definir constantes ➡ `DEFAULT_CAPACITY` de `Vector`

Método estáticos:

- ♦ Pueden utilizar los atributos de clase pero no los de instancia
- ♦ Sintaxis: Hay que cualificarlos con el nombre de la clase

```
NombreClase::metodoStatic(argumentos);
```

Atributos y Métodos de Clase (static)

- ♦ Ejemplo: Contador de nº de objetos de una clase y recurso compartido

```
class CC {  
private:  
    static int contCC; // Contador de objetos de la clase CC  
    static Date* pRec; // Compartido por todos los objetos de la clase  
public:  
    static const int MAX_CC = 10; // una constante y su valor  
    static int getContCC() { // Para consultar el contador  
        return contCC;  
    }  
    CC();  
    ~CC();  
};
```

CC.h

Atributos y Métodos de Clase (static)

- ♦ Ejemplo: Contador de n° de objetos de una clase y recurso compartido

```
#include "CC.h"
```

CC.cpp

```
XXX* CC::pRec = nullptr; // Valor inicial obligatorio  
int CC::contCC = 0;      // Valor inicial obligatorio
```

```
CC::CC() {  
    if (pRec == nullptr)  
        pRec = new XXX; // constructora de XXX  
    ++contCC;  
}
```

```
CC::~~CC() {  
    --contCC;  
    if (contCC == 0) {  
        delete pRec;  
        pRec = nullptr;  
    }  
}
```

Excepciones y POO

- ✦ Aunque en C++ no es obligatorio, lo recomendable es que las excepciones sean objetos
- ✦ Cualquier objeto puede ser excepción (no se le exige nada)
- ✦ Ejemplo:

```
class Error {  
protected:  
    string mensaje;  
public:  
    Error(string const& m) : mensaje(m) {};  
    const string& getMensaje() const {  
        return mensaje;  
    };  
};
```

- ✦ También es recomendable y habitual organizar los tipos de excepciones en jerarquías de excepciones, tanto las definidas por el programador como las de librerías. Lo veremos en el Tema 4

Excepciones y P00

- ♦ Se pueden lanzar los punteros a objetos o los objetos in situ
- ♦ Al recibir el objeto en el catch, si se lanzó el objeto in situ, es recomendable hacerlo por referencia para evitar la copia

```
template<class T>
void Vector<T>::pop_back(){
    if (count == 0) throw Error("Empty vector exception");
    --count;
}
```

```
Vector<int> v;
...
try {
    v.pop_back();
} catch (Error& e){
    cout << e.what() << endl;
}
```