Tema 2: Programación orientada a objetos con C++

Tecnología de la Programación de Videojuegos 1

Grado en Desarrollo de Videojuegos

Curso 2023-2024

Miguel Gómez-Zamalloa Gil con cambios de Rubén Rubio Cuéllar

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: youtu.be/HNMAXCfP6K4?t=3540
- 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, por copia y por movimiento
    - Operador de asignación
  - Destructoras
- Distribución del código en ficheros .h y .cpp
- Genericidad mediante plantillas
- Más diferencias:
  - El `main` no es método de una clase
  - En C++ no hay interfaces
  - Herencia múltiple
  - Etc.

Interfaz vs. implementación

Interfaz

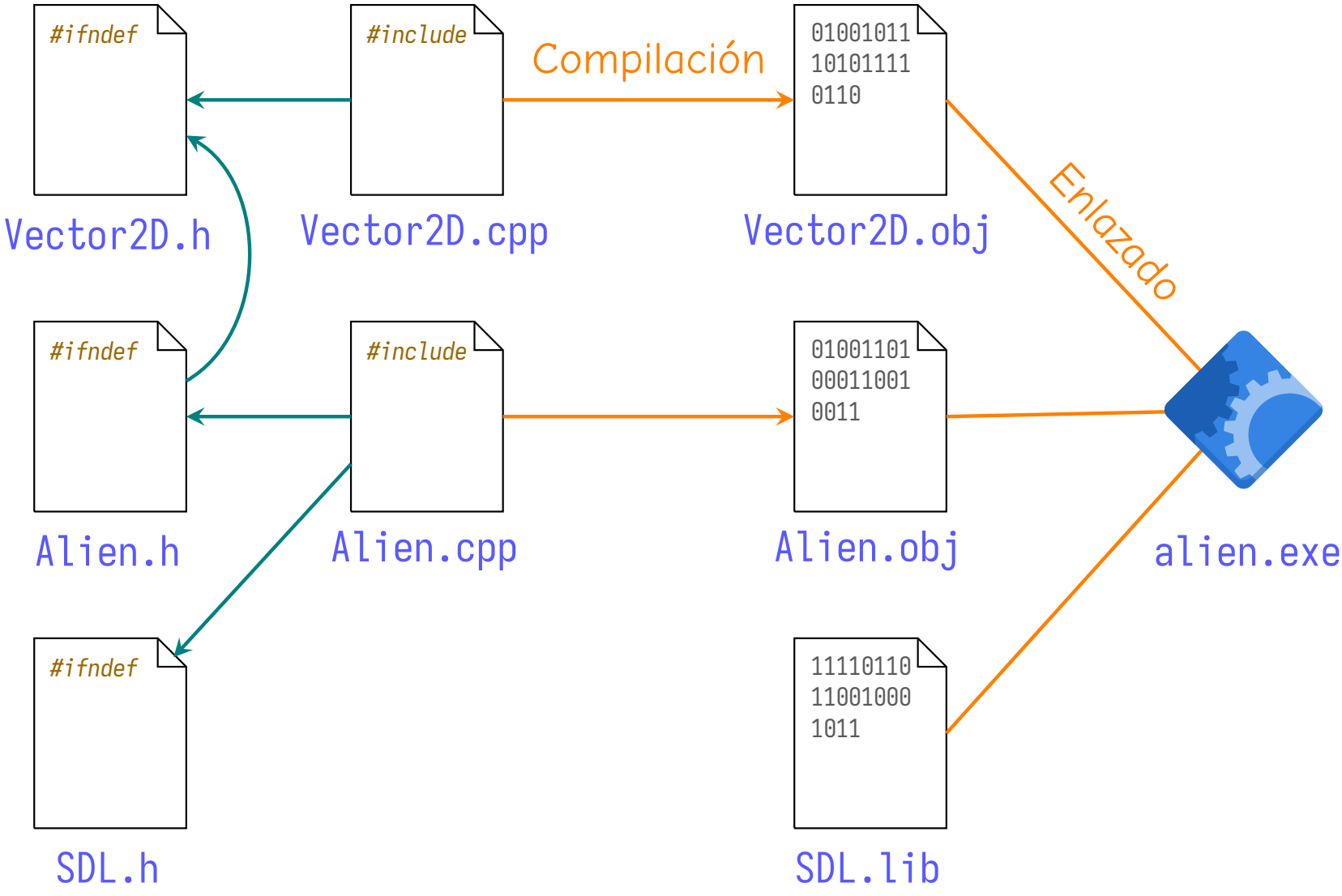
- Todo lo que el usuario de la clase necesita saber
  - Tipos de datos
  - Declaraciones de métodos y/o funciones
  - Declaración de constantes

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 lo 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

Interfaz vs. implementación



(Interfaz vs. implementación)

Módulos en C++20 (anécdota): alternativa a la separación .cpp/.h para evitar algunos problemas asociados a las cabeceras (derivados de que *#include* y *#define* son sustituciones sintácticas).

holamundo.cpp

```
export module holamundo;
import <iostream>;

export void holaMundo()
{
    std::cout << "¡Hola Mundo!\n";
}
```

main.cpp

```
import holamundo;

int main()
{
    holaMundo();
}
```

No los usaremos, pues no están implantados en la práctica habitual y el soporte de la mayoría de compiladores no es completo.

Ejemplo: Vector2D

Vector2D.h

```
#ifndef VECTOR2D_H_    // Normalmente lo escriben los IDEs,
#define VECTOR2D_H_    // VS escribe #pragma once
#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

```cpp
#include "Vector2D.h"
#include <cmath>
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, solo se ilustra 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

```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

VectorOfDoubles.h

```
#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 size_t = unsigned long int; // Alias por comodidad (ya definido en la STL)

class VectorOfDoubles {
private:
   static constexpr size_t DEFAULT_CAPACITY = 5; // Capacidad inicial por defecto
   size_t capacity; // Capacidad actual del array dinámico
   size_t 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; } // Ver P14
   size_t 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;}
   ...
```

Ejemplo: VectorOfDoubles

VectorOfDoubles.h

```
    ...
    void push_back(double e); // Pone nuevo elemento al final
    void pop_back(); // Quita último elemento
    bool insert(double e, size_t i); // Inserta e en posición iésima (desplazando)
    bool erase(size_t i); // Borra el elemento de la pos. iésima (desplazando)

private:
    void reallocate();
    void shiftRightFrom(size_t i);
    void shiftLeftFrom(size_t i);
};

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

Ejemplo: VectorOfDoubles

VectorOfDoubles.cpp

```
#include "VectorOfDoubles.h"
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, size_t 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(size_t i) {
   if (i ≥ numElems) return false;
   else {
      shiftLeftFrom(i);
      --numElems;
      return true;
   }
}
// Métodos privados
void VectorOfDoubles::shiftRightFrom(size_t i) {
   for (size_t j = numElems; j > i; j--)
      elems[j] = elems[j-1];
}
void VectorOfDoubles::shiftLeftFrom(size_t 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 (size_t i = 0; i < size(); ++i)
      newElems[i] = elems[i];
   delete[] elems;
   elems = newElems;
}
```

Destructora

- Toda clase tiene un método especial denominado destructora 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
- 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;
public:
   A(int i) : i(i) {
       cout << "ctor a" << i << '\n';
   }

   ~A() {
       cout << "dtor 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
dtor a2
dtor a3
dtor a1
dtor 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) o incluso 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 más en profundidad en EDA

Ejemplo: vector genérico

Vector.h

```
#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 <typename T> // o template <class T>
class Vector {
private:
   static constexpr size_t DEFAULT_CAPACITY = 5; // Capacidad inicial por defecto
   size_t capacity; // Capacidad actual del array dinámico
   size_t 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;}
   size_t 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, size_t 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.h

```
Vector<Vector2D> v; /* Instanciación (como 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];
```

Constructor y asignación por movimiento

- El constructor por movimiento "roba" o transfiere los recursos de `obj` al nuevo objeto (o al objeto asignado en la asignación)

```
Clase(Clase &&obj) { ... }
Clase& operator=(Clase &&obj) { ... }
```

- El objeto `obj` debe quedar en un estado válido pero indeterminado
- Se utiliza en los **`return`** o con `std::move(obj)` en asignaciones o llamadas a función

```
Vector<T>::Vector(Vector<T> &&obj)
 : capacity(obj.capacity),
   numElems(obj.numElems),
   elems(obj.elems)
{   obj.numElems = obj.capacity = 0;
    obj.elems = nullptr;
}
```

```
Vector<T> generaVector() {
   Vector w; // [...] lo rellena
   return w;
}
int main() { /* ... */
   Vector v = generaVector(); /* ... */
}
```

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 inicializarlos
- 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

Ave.h

```
class Ave {
private:
   static int contador;   // Contador de objetos de la clase Ave
   static Vector2D* dir;  // Compartido por todos los objetos Ave
public:
   static constexpr int VX_INI = 10, VY_INI = 0;  // constantes

   static int getContador() { // Para consultar el contador
      return contador;
   }

   Ave();
   ~Ave();
};
```

Atributos y métodos de clase (static)

Ejemplo: contador de nº de objetos de una clase y recurso compartido

Ave.cpp

```
#include "Ave.h"
Vector2D* Ave::dir = nullptr;  // Valor inicial obligatorio
int Ave::contador = 0;         // Valor inicial obligatorio

Ave::Ave() {
   if (dir == nullptr)  // Construcción de recurso compartido
      dir = new Vector2D(Ave::VX_INI, VY_INI);
   ++contador;
}
Ave::~Ave() {
   if (--contador == 0) {
      delete dir;
      dir = 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& what() const {
      return mensaje;
   };
};
```

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

Excepciones y POO

- 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;
}
```