Programación Orientada a Objetos

Unidad 2: Introducción a la P.O.O.

Pablo Novara

2024

UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

FICH UNL

¿QUÉ ES LA POO?

Problema Real

simplificar la gestión de la universidad

Interfaz de bajo nivel

int *
char [256]
*(x+i)
...

MECANISMOS DE ABSTRACCIÓN

> *Controlling complexity is the essence of computer programming.*

Brian Kernighan

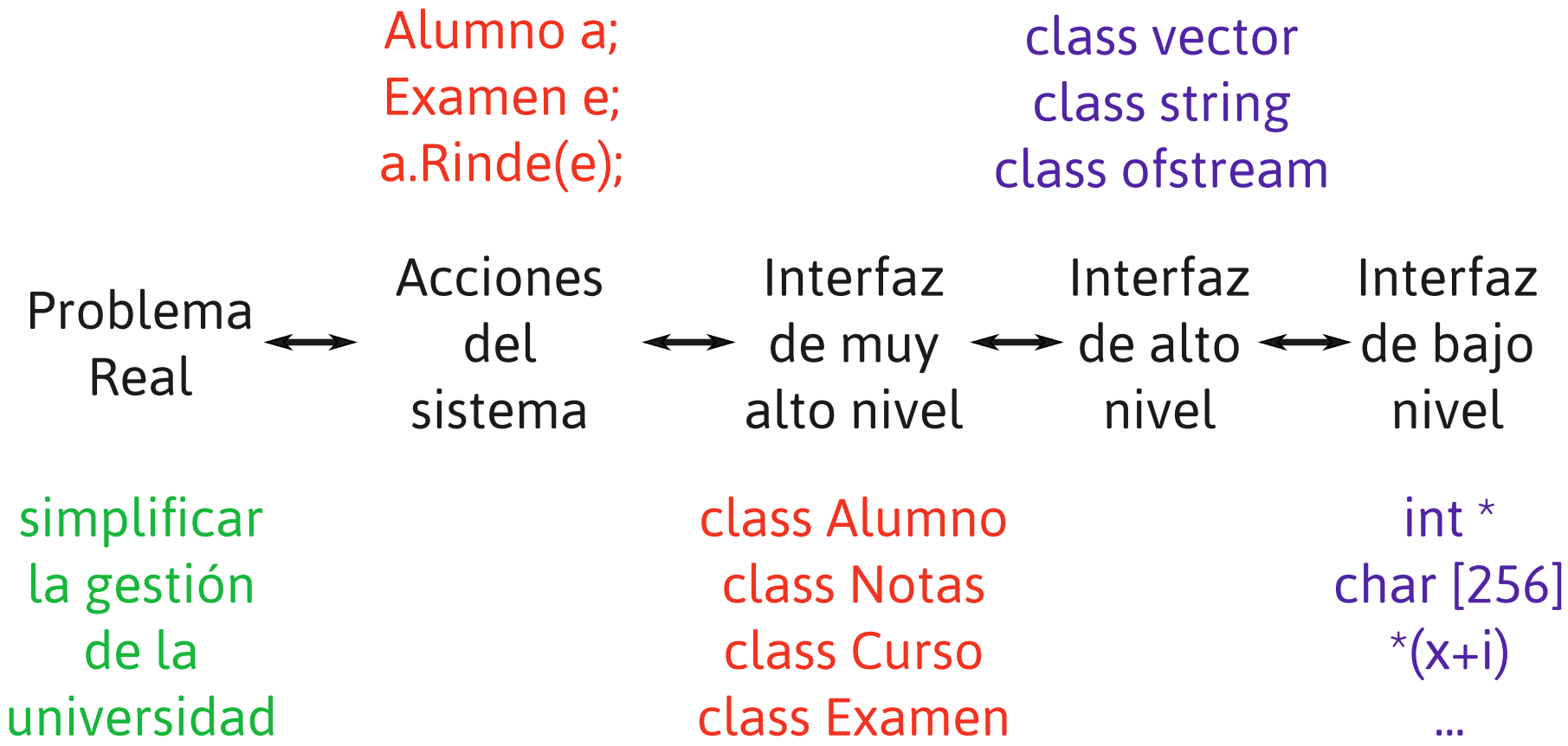
¿Qué mecanismos de abstracción/reducción de la complejidad conocen (hasta ahora)?

¿QUÉ ES LA POO?

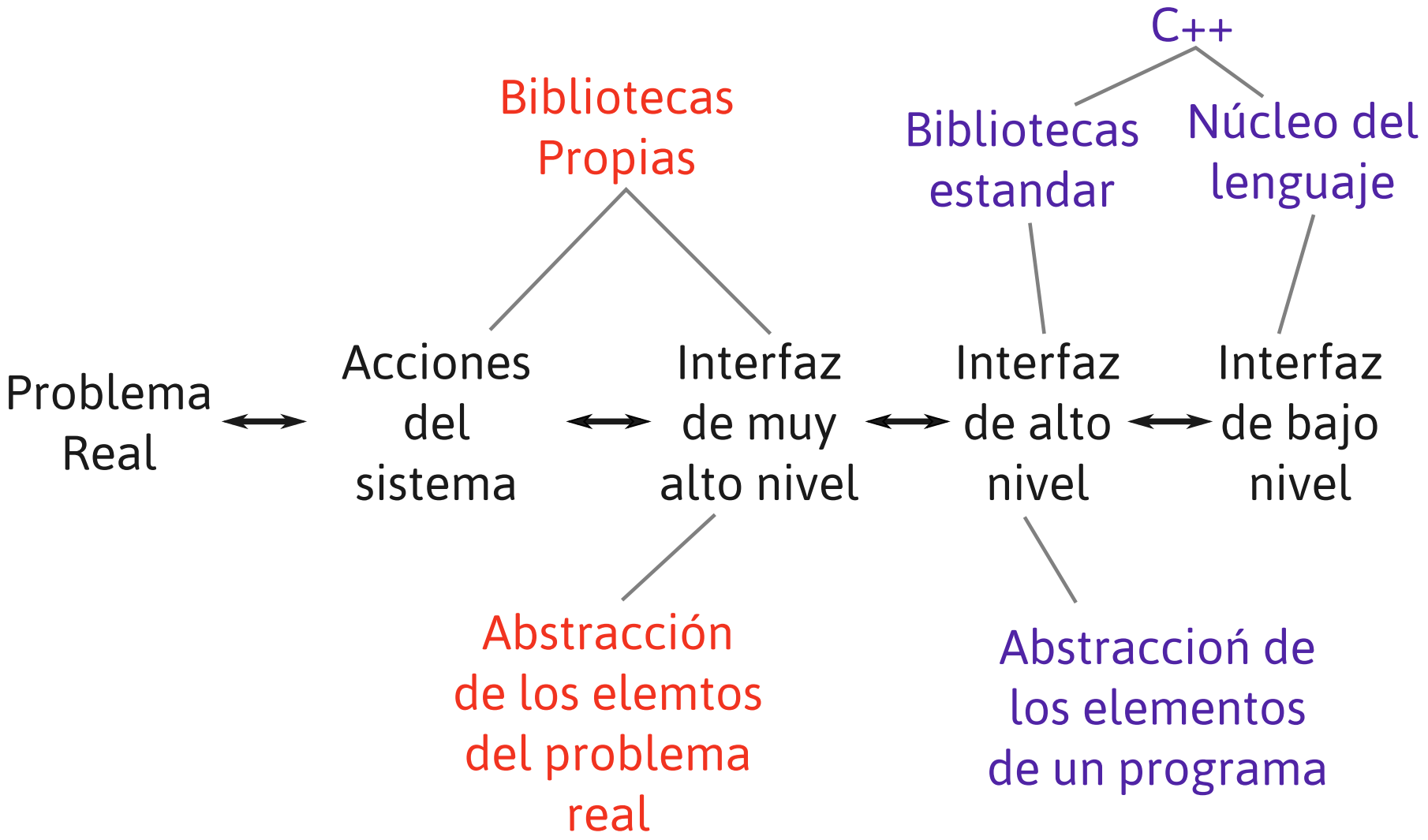
La Programación Orientada a Objetos es un paradigma que utiliza objetos como elementos fundamentales en la construcción de la solución.

El objetivo es describir el problema y plantear la solución en los términos del problema mismo y no de elementos computacionales

¿QUÉ ES LA POO?


Alumno a;
Examen e;
a.Rinde(e);
class vector
class string
class ofstream
Problema Real
Acciones del sistema
Interfaz de muy alto nivel
Interfaz de alto nivel
Interfaz de bajo nivel
simplificar la gestión de la universidad
class Alumno
class Notas
class Curso
class Examen
int *
char [256]
*(x+i)
...

¿QUÉ ES LA POO?


C++
Bibliotecas Propias
Bibliotecas estandar
Núcleo del lenguaje
Problema Real
Acciones del sistema
Interfaz de muy alto nivel
Interfaz de alto nivel
Interfaz de bajo nivel
Abstracción de los elemtos del problema real
Abstracción de los elementos de un programa

¿QUÉ SON LOS OBJETOS?

Objeto: entidad provista de un conjunto de datos o estado, y un conjuto de funcionalidades o comportamiento

En C++:
datos = atributos o variables miembro
comportamiento = métodos o funciones miembro

Clase: definición de las propiedades y comportamientos de un tipo de objeto concreto.

Instanciación: creación de un objeto a partir de la definición de una clase.

OBJETOS EN C++

```
class <nombre de la clase> {

    public:

        lo que sí se ve desde afuera del objeto


    private:

        lo que no se ve desde afuera del objeto


};
```

PRINCIPIO DE OCULTACIÓN

Cada objeto está aislado del exterior, y
expone solo una interfaz que
especifica cómo puede interactuar.

El aislamiento protege a las propiedades
internas de un objeto,
garantizando su integridad (*invariantes*).

En la mayoría de los casos, todos los atributos de un objeto deberían ser privados, exponiendo solamente métodos públicos al exterior.

OBJETOS EN C++

```
class <nombre de la clase> {

    public:
        <métodos públicos>
            interfaz del objeto
        <atributos públicos>
                mala idea

    private:
        <atributos privados>
            estado del objeto
        <métodos privados>
            funciones auxiliares

};
```

UTILIZACIÓN DE OBJETOS

```
float x, y, z;
cin >> x >> y >> z;
// se crea el objeto y se definen sus atributos iniciales
Ecuacion eq;
eq.CargarCoefs(x,y,z);
// se opera con el objeto y/o consulta su estado
if (eq.TieneRaicesReales()) {
  cout << "r1=" << eq.VerRaiz1() << endl;
  cout << "r2=" << eq.VerRaiz2() << endl;
} else {
  cout << "r1=" << eq.VerParteReal() << "+"
       << eq.VerParteImag() << "i" << endl;
  cout << "r1=" << eq.VerParteReal() << "-"
       << eq.VerParteImag() << "i" << endl;
}
```

UTILIZACIÓN DE OBJETOS

```
float x, y, z;
cin >> x >> y >> z;
// se crea el objeto y se definen sus atributos iniciales
Ecuacion eq;
eq.CargarCoefs(x,y,z);
// se opera con el objeto y/o consulta su estado
if (eq.TieneRaicesReales()) {
  cout << "r1=" << eq.VerRaiz1() << endl;
  cout << "r2=" << eq.VerRaiz2() << endl;
} else {
  cout << "r1=" << eq.VerParteReal() << "+"
       << eq.VerParteImag() << "i" << endl;
  cout << "r1=" << eq.VerParteReal() << "-"
       << eq.VerParteImag() << "i" << endl;
}
```

*No conviene usar **cin**/**cout** dentro de la clase*

UTILIZACIÓN DE OBJETOS

```
float x, y, z;
cin >> x >> y >> z;
// se crea el objeto y se definen sus atributos iniciales
Ecuacion eq;
eq.CargarCoefs(x,y,z);
// se opera con el objeto y/o consulta su estado
if (eq.TieneRaicesReales()) {
  cout << "r1=" << eq.VerRaiz1() << endl;
  cout << "r2=" << eq.VerRaiz2() << endl;
} else {
  cout << "r1=" << eq.VerParteReal() << "+"
       << eq.VerParteImag() << "i" << endl;
  cout << "r1=" << eq.VerParteReal() << "-"
       << eq.VerParteImag() << "i" << endl;
}
```

*Los datos se le pasan al objeto **solo una vez**, el objeto los **"recuerda"***

DEFINICIÓN DE CLASES

```
class Ecuacion {
public:
     // interfaz para carga de datos
    void CargarCoefs(float a, float b, float c);
     // interfaz para consulta de resultados
    bool TieneRaicesReales();
    float CalcularRaiz1();        // si son reales
    float CalcularRaiz2();
    float CalcularParteReal();    // si son complejas
    float CalcularParteImag();
private:
     // atributos cargados por el usuario
    float m_a, m_b, m_c;
     // método auxiliar para resolver los cálculos
    float Discriminante();
};
```

IMPLEMENTACIÓN DE MÉTODOS

Un método implementa **dentro** de la clase como una función...

```
class Ecuacion {
public:
    ...
    void CargarCoefs(float a, float b,  float c) {
        m_a = a; m_b = b; m_c = c;
    }
    ...
private:
    float m_a, m_b, m_c;
    ...
};
```

✅ *Dentro del método, se puede acceder directamente a los atributos y demás métodos de esa clase.*

IMPLEMENTACIÓN DE MÉTODOS

...o **fuera** de la clase agregando el scope (nombre de la clase) al nombre de la función/método:

```
class Ecuacion {
public:
    ...
    void CargarCoefs(float a, float b, float c); // solo prototipo
    ...
};
```

```
void Ecuacion::CargarCoefs(float a, float b, float c) {
    m_a = a; m_b = b; m_c = c;
}
```

✅ *esta implementación podría estar en otro archivo fuente diferente*

IMPLEMENTACIÓN DE MÉTODOS

Evitar errores comunes:

```
float Ecuacion::Discriminante() {
    return m_b*m_b-4*m_a*m_c;
}
```

***Este método no necesitan recibir nada**, los datos de entrada ya estarán en los atributos gracias al método CargarCoefs*

```
float Ecuacion::VerRaiz1() {
    float d = Discriminante();
    return ( -m_b + sqrt(d) ) / ( 2 * m_a );
}
```

*las variables auxiliares de un método **no son atributos** de la clase*

EJEMPLOS DE CLASES/OBJETOS

- Cosas tangibles: **Auto, Arma, Disparo**
- Roles o papeles: **Alumno, Empleado**
- Organizaciones/Sistema: **Empresa, Universidad, Juego**
- Incidentes o sucesos: **Liquidación, Examen**
- Interacciones o relaciones: **Pedido, Alquiler**
- Abstracciones de más bajo nivel: **Vector, Matriz, Archivo**

EJEMPLOS DE MÉTODOS

- Definir/modificar el estado:
  - void **Alumno::ActualizarEMail**(string nueva_dir_correo);
  - void **Personaje::SumarPuntos**(int pts_ganados);
  - void **Tecla::Apretar**(); void **Tecla::Soltar**();
  - void **Curso::InscribirAlumno**(string nombre);
- Consultar el estado
  - string **Alumno::VerTelefono**();
  - int **Personaje::VerVidas**();
  - bool **Tecla::EstaApretada**();
  - int **Curso::CantidadDeInscriptos**();
  - string **Curso::NombreAlumno**(int i);

EJEMPLOS DE MÉTODOS (CONT.)

- Calcular cosas nuevas
  - float **Alumno::ObtenerPromedio**();
  - int **Curso::MejorAlumno**();
  - int **Persona::CalcularEdad**(int fecha_actual);
  - float **Polinomio::Evaluar**(float x);

- Constructor/Destructor: dentro de 2 slides

Diseñar la interfaz pública pensando desde el punto de vista del programa cliente (como caja negra).

EJEMPLOS DE ATRIBUTOS

- En clase Alumno:
  - nombre y apellido
  - dni, email, teléfono
  - vector de notas **ó** promedio
  - ~~edad~~ fecha_nacimiento

no ambas, información redundante

edad se desactualiza sola

Poner como atributo lo mínimo que necesite para dar soporte a las funcionalidades de la clase.

CONSTRUCTORES Y DESTRUCTORES

Constructor:

- método especial que **se invoca automáticamente al crear un objeto**

¿Cuándo ocurre eso?

- tiene el mismo nombre que la clase
- puede recibir argumentos y sobrecargarse
- si no se especifica ninguno, c++ otorga dos por defecto:
  - **constructor nulo**: no hace "nada" (también llamado "por defecto")
  - **constructor de copia**: copia uno por uno los atributos desde otro objeto de la misma clase

CONSTRUCTORES Y DESTRUCTORES

Destructor:

- método que **se invoca automáticamente al destruir un objeto**

¿Cuándo ocurre eso?

- tiene por nombre el caracter **~** más el nombre que la clase
- no puede recibir parámetros
- por defecto no hace "nada"

CONSTRUCTORES Y DESTRUCTORES POR DEFECTO

Los constructores y destructores generados por el compilador equivalen *aproximadamente* a:

```
class Ecuacion {
public:
    Ecuacion() { /* naaada */ }
    Ecuacion(const Ecuacion &o) {
        /** copia atributo por atributo **/
        m_a=o.m_a; m_b=o.m_b; m_c=o.m_c;
    }
    ~Ecuacion() { /* naaada */ }
    // ... varios metodos públicos...
private
    float m_a, m_b, m_c;
};
```

CONSTRUCTORES Y DESTRUCTORES

```
class Ecuacion {
    ...
public:
    Ecuacion(float a, float b, float c);
    ...
};
```

Al explicitar un constructor, se deshabilita el nulo que C++ generaba por defecto

```
int main() {
    float x, y, z;
    cin >> x >> y >> z;
    Ecuacion eq(x, y, z);
    if (eq.TieneRaicesReales()) {
        ...
```

Ahora es obligación pasar tres floats para crear una Ecuacion. Es imposible obtener una ecuación con coefs. sin inicializar.

EJEMPLO RAII: MEMORIA

```
class Vector {
public:
    Vector(int n) { m_data = new int[n]; } // adquisición
    ~Vector() { delete [] m_data; } // liberación
    // ...metodos varios que usan la memoria apuntada por m_data...
private:
    int *m_data;
};
```

```
void foo() {
    Vector v1(10); // adquisición inevitable
    // ...uso del vector v1...
} // liberación automática
```

> *Just that closing brace. Here is where all the magic happens.*
>
> *Bjarne Stroustroup*

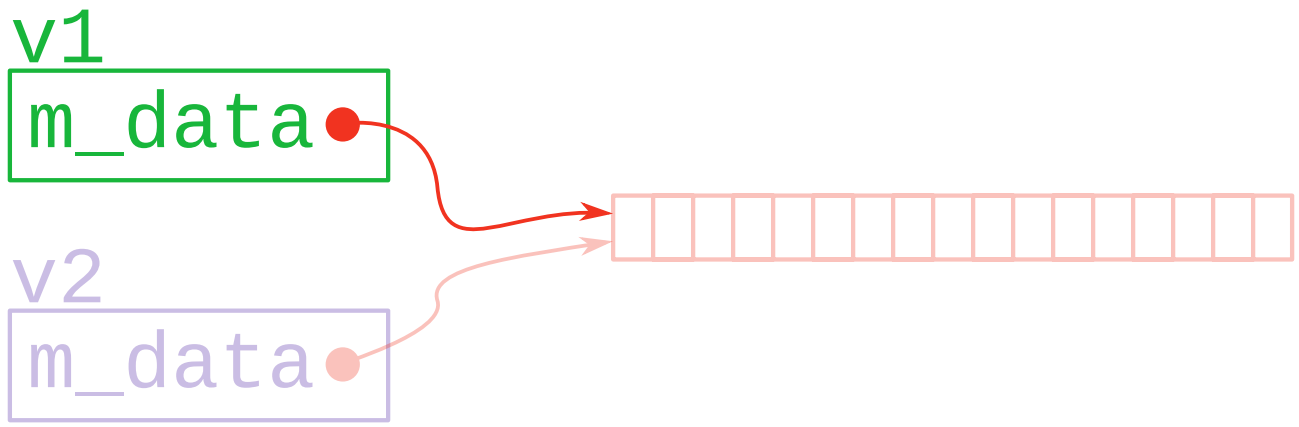
FILOSOFÍA RAII

Para todo **recurso** que se deba *adquirir* y *liberar*:

- Se utiliza un objeto que represente el recurso
- El objeto debe garantizar la correcta utilización del mismo
  - Los **constructores** garantizan la adquisición previa al uso
  - El **destructor** garantiza la liberación luego del uso
  - La ocultación garantiza que nadie lo "rompa" durante el uso.
- Ejemplos de recursos:
  - archivos, conexiones de red, memoria, ...

CONSTRUCTOR DE COPIA

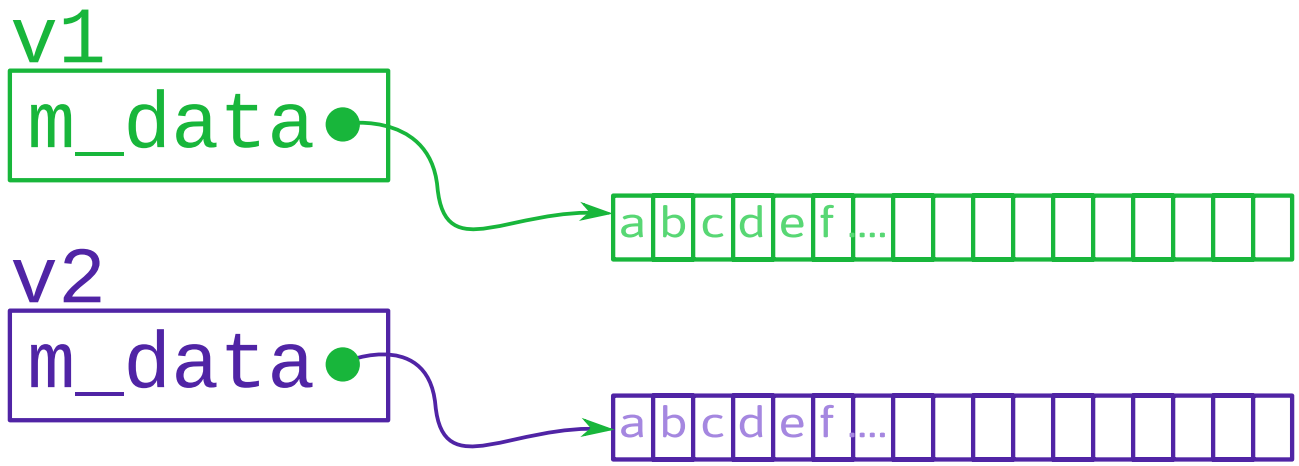
```
int main() {
    Vector v1(100);
    ...
    Vector v2 = v1; // copia, equivale a v2(v1)
    ...
} // error: doble delete
```



*Si una clase gestiona un recurso **es necesario** rehacer (o prohibir?) el constructor de copia*

CONSTRUCTOR DE COPIA

```
int main() {
    Vector v1(100);
    ...
    Vector v2 = v1; // copia, equivale a v2(v1)
    ...
}
```



CONSTRUCTOR DE COPIA

```
class Vector {
  int *m_data;
  int m_n;
public:
  Vector(int n) { // constructor usual
    m_data = new int[n];
    m_n = n;
  }

  Vector(const Vector &v2) { // constructor de copia
    // obtener memoria para otro vector
    m_n = v2.m_n;
    m_data = new int[m_n];
    // copiar los datos de un vector en otro
    for(int i=0;i<m_n;i++)
      m_data[i] = v2.m_data[i] ;
  }
```

CONSTRUCTOR DE "MOVE"

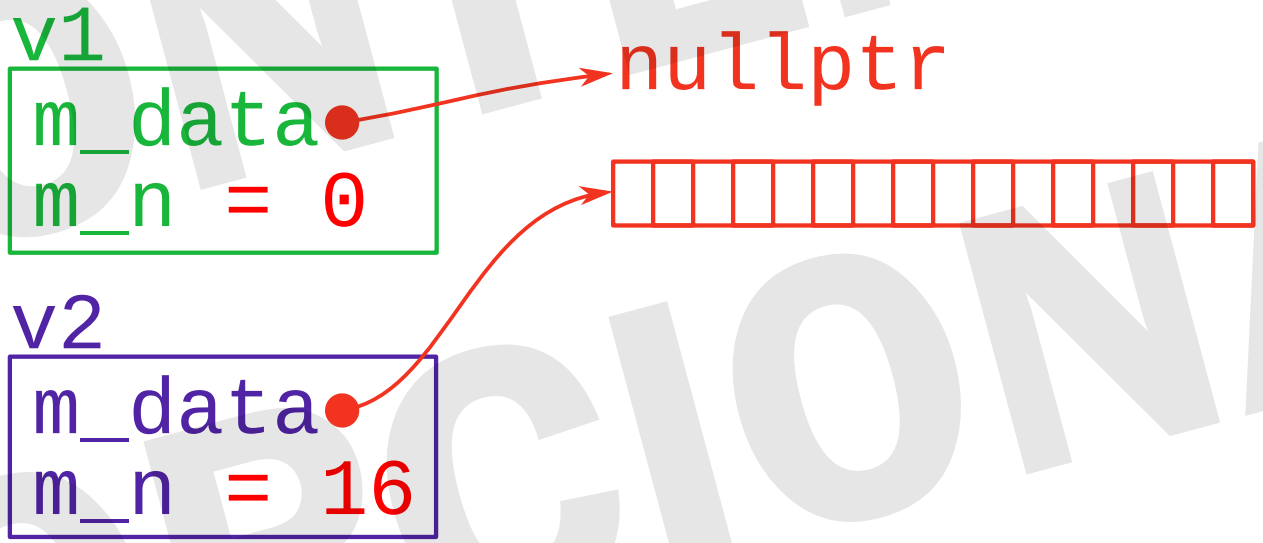
```
class Vector {
  int *m_data;
  int m_n;
public:
  Vector(int n) { ... }
  Vector(const Vector &v2) { ... }

  Vector(Vector &&v2) { // move-ctor
    // robarle la memoria a v2
    m_n = v2.m_n;
    m_data = v2.m_data;
    // dejar a v2 en un estado válido
    v2.m_data = nullptr;
    v2.m_n = 0;
  }
```

MOVE-SEMANTICS

```
Vector foo(int n) { ... }

int main() {
    Vector v2 = foo(100);
    ...
```



EL PUNTERO this

Dentro de un método, **this** representa un puntero al objeto mediante el cual se invocó a dicho método.

```
class Mascota {
    string m_nombre;
    ...
    string VerNombre() {
        return m_nombre;
    }
    ...
};
void Foo() {
    Mascota loro("Polly"), caracol("Turbo");
    cout << loro.VerNombre() << endl;
    cout << caracol.VerNombre() << endl;
}
```

EL PUNTERO this

```
class Mascota {
    string m_nombre;
    ...
    string VerNombre() {
        return m_nombre;
                equivale a this->m_nombre
    }
    ...
};

void Foo() {
    Mascota loro("Polly"), caracol("Turbo");
    cout << loro.VerNombre() << endl;
                this toma &loro
    cout << caracol.VerNombre() << endl;
                this toma &caracol
}
```

EL PUNTERO this

```
class Calculadora {
  double num;
public:
  Calculadora();
  void Sum (double num);
  void Rest(double num);
  void Mult(double num);
  void Div (double num);
  double Result();
};

int main() {
  Calculadora calc;
  calc.Sum(9);
  calc.Rest(3);
  calc.Mult(7);
  cout << calc.Result(); // muestra 42
}
```

EL PUNTERO this

```
class Calculadora {
  double num;
public:
  Calculadora() { this->num = 0.0; }
  void Sum (double num) { this->num += num; }
  void Rest(double num) { this->num -= num; }
  void Mult(double num) { this->num *= num; }
  void Div (double num) { this->num /= num; }
  double Result() { return this->num; }
};

int main() {
  Calculadora calc;
  calc.Sum(9);
  calc.Rest(3);
  calc.Mult(7);
  cout << calc.Result(); // muestra 42
}
```

EL PUNTERO this

```
class Calculadora {
  double num;
public:
  Calculadora() { this->num = 0.0; }
  Calculadora &Sum(double num) {
    this->num += num; return *this;
  }
  Calculadora &Rest(double num) {
    this->num -= num; return *this;
  }
  Calculadora &Mult(double num) {
    this->num *= num; return *this;
  }
  Calculadora &Div(double num) {
    this->num /= num; return *this;
  }
  double Result() { return this->num; }
};
```

EL PUNTERO this

```
class Calculadora {
  double num;
public:
  Calculadora() { this->num = 0.0; }
  Calculadora &Sum(double num) { ... }
  Calculadora &Rest(double num) { ... }
  Calculadora &Mult(double num) { ... }
  Calculadora &Div(double num) { ... }
  double Result() { return num; }
};

int main() {
  Calculadora calc;
  cout<<calc.Sum(9).Rest(3).Mult(7).Result();
}
```

MÉTODOS const

```
class Calculadora {
  ...
public:
  ...
  double Result();
};

void MostrarResultado(const Calculadora &c) {
  cout<<"El resultado es: "<<c.Result()<<endl;

}

int main() {
  Calculadora calc;
  calc.Sum(9).Rest(3).Mult(7);
  MostrarResultado(calc);
}

double Calculadora::Result() { return num; }
```

No compila!

MÉTODOS const

```
class Calculadora {
  ...
public:
  ...
  double Result() const;
};

void MostrarResultado(const Calculadora &c) {
  cout<<"El resultado es: "<<c.Result()<<endl;
}

int main() {
  Calculadora calc;
  calc.Sum(9).Rest(3).Mult(7);
  MostrarResultado(calc);
}

double Calculadora::Result() const { return num; }
```

Ok, el método es "const"

INICIALIZACIÓN DE MIEMBROS

1. Asignar dentro del constructor

```
class Calculadora {
  double num;
public:
  Calculadora() { num = 0.0; }
  ...
};
```

✅ *Esta es la opción más "simple" para empezar*

INICIALIZACIÓN DE MIEMBROS

2. Asignar en la definición de la variable miembro

```
class Calculadora {
  double num = 0.0;
public:
  Calculadora() { /*nada*/ }
  ...
};
```

En este caso no haría falta ni declarar el constructor

Solo se simple con tipos de datos simple (no sirve por ej. para el tamaño de un vector)

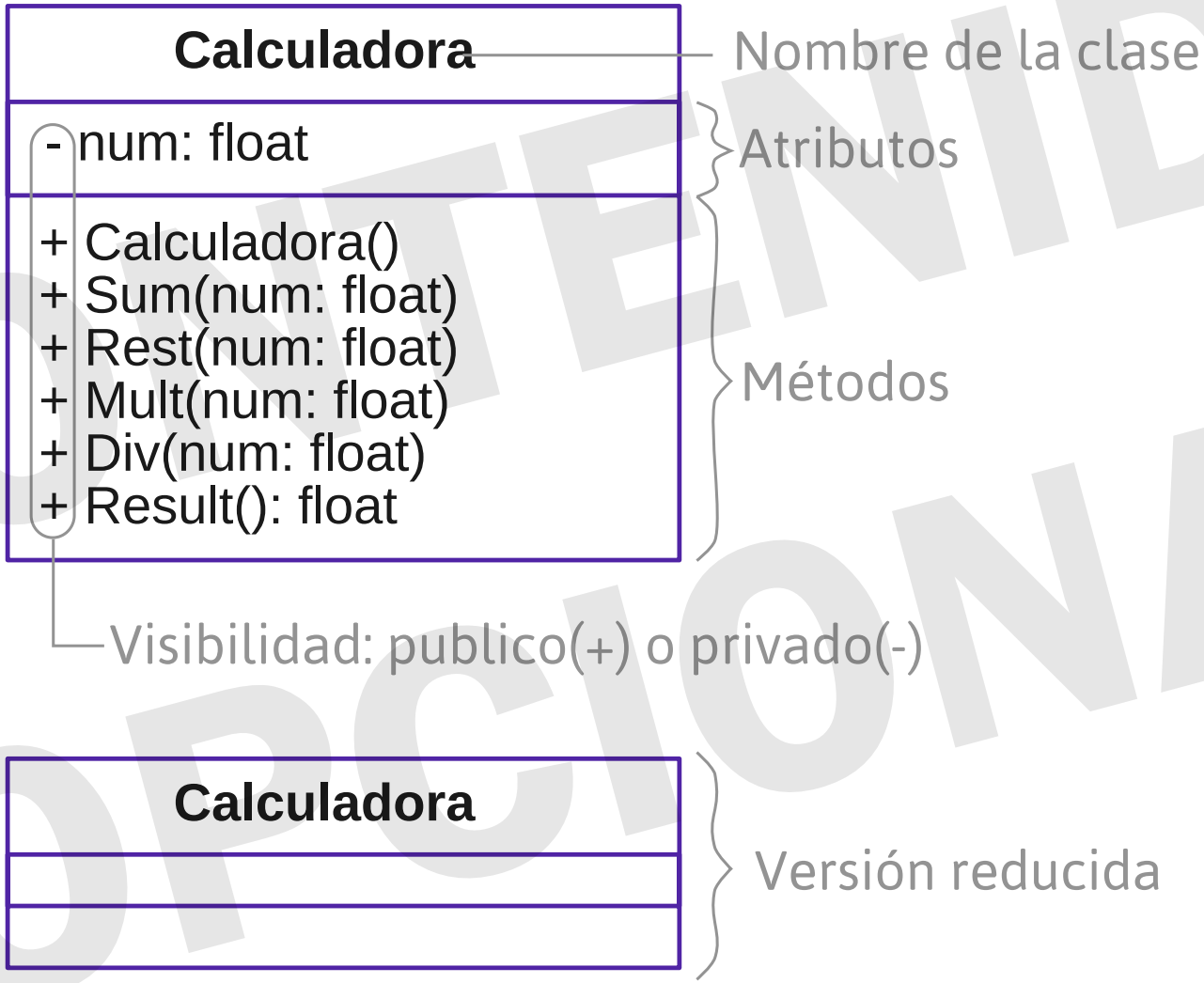
INICIALIZACIÓN DE MIEMBROS

3. Sintaxis especial para inicialización de variables miembro

```
class Calculadora {
  double num;
public:
  Calculadora() : num(0.0) { /*nada*/ }
  ...
};
```

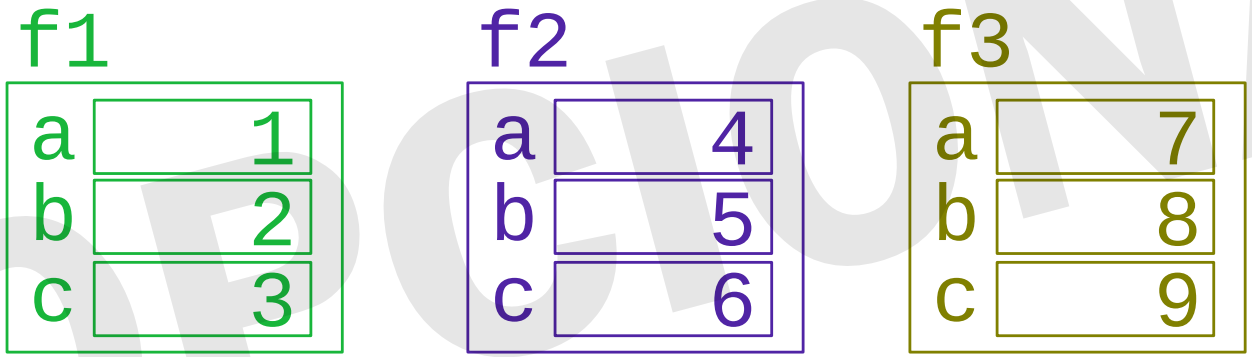
✅ *Esta es la opción funciona en todos los casos (y en la unidad 3 a veces va a ser la única)*

NOTACIÓN UML



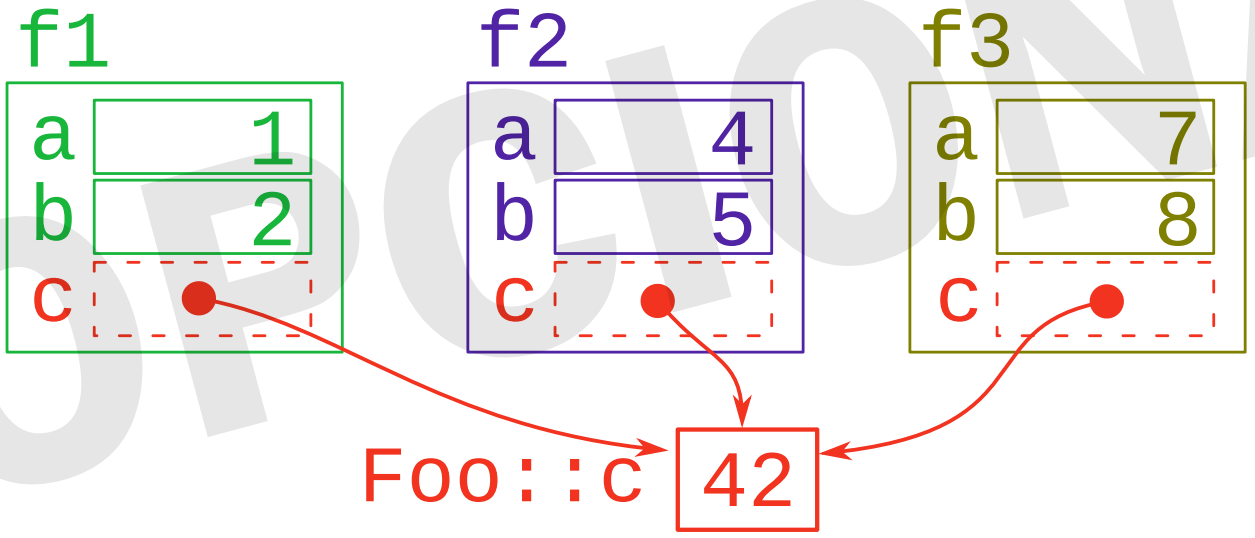
ATRIBUTOS Y MÉTODOS static

```
class Foo {
    int a;
    int b;
    int c;
public:
    ...
};
int main() {
    Foo f1, f2, f3;
```



ATRIBUTOS Y MÉTODOS static

```
class Foo {
    int a;
    int b;
    static int c;
public:
    ...
};
int main() {
    Foo f1, f2, f3;
```

ATRIBUTOS Y MÉTODOS `static`

```
class Foo {
    int a;
    int b;
    static int c;
public:
    static int VerC() {
        return c;
    }
    ...
};
int main() {
    cout << Foo::VerC();
```

Un método `static` solo puede acceder a atributos `static`