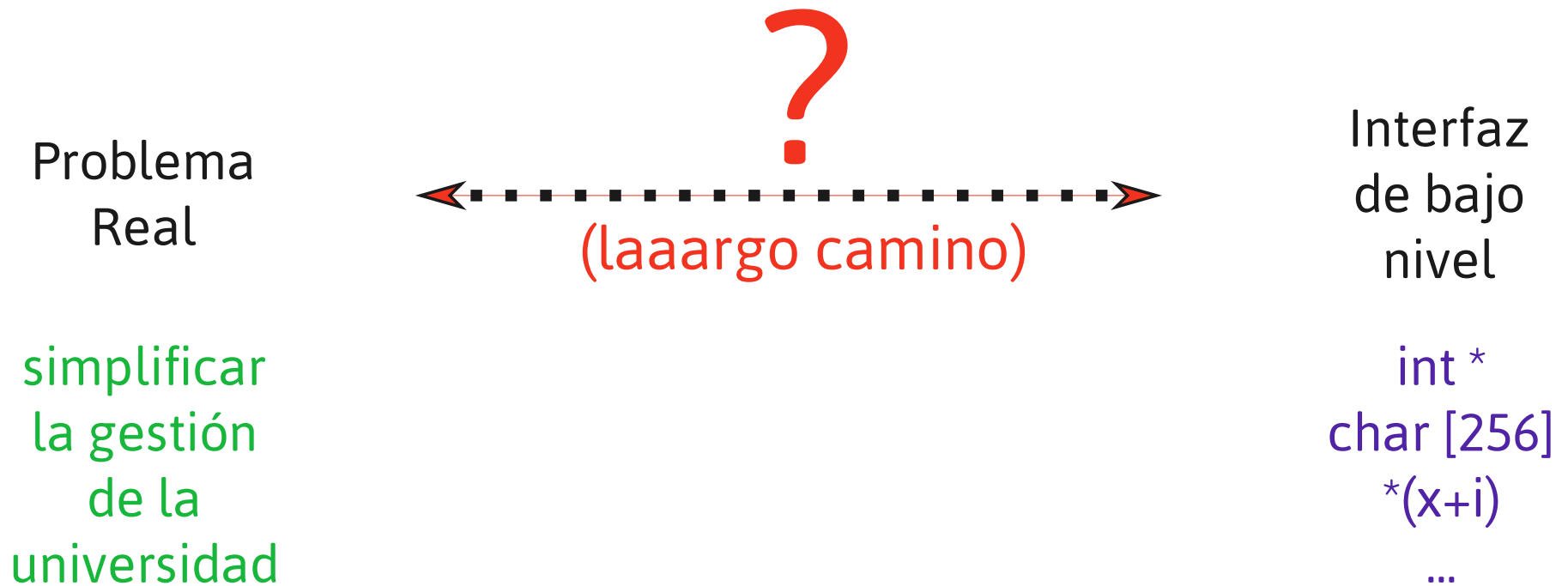


Programación Orientada a Objetos

Unidad 2: Introducción a la P.O.O.

¿QUÉ ES LA POO?



MECANISMOS DE ABSTRACCIÓN

“Controlling complexity is the essence of computer programming.”

Brian Kernighan

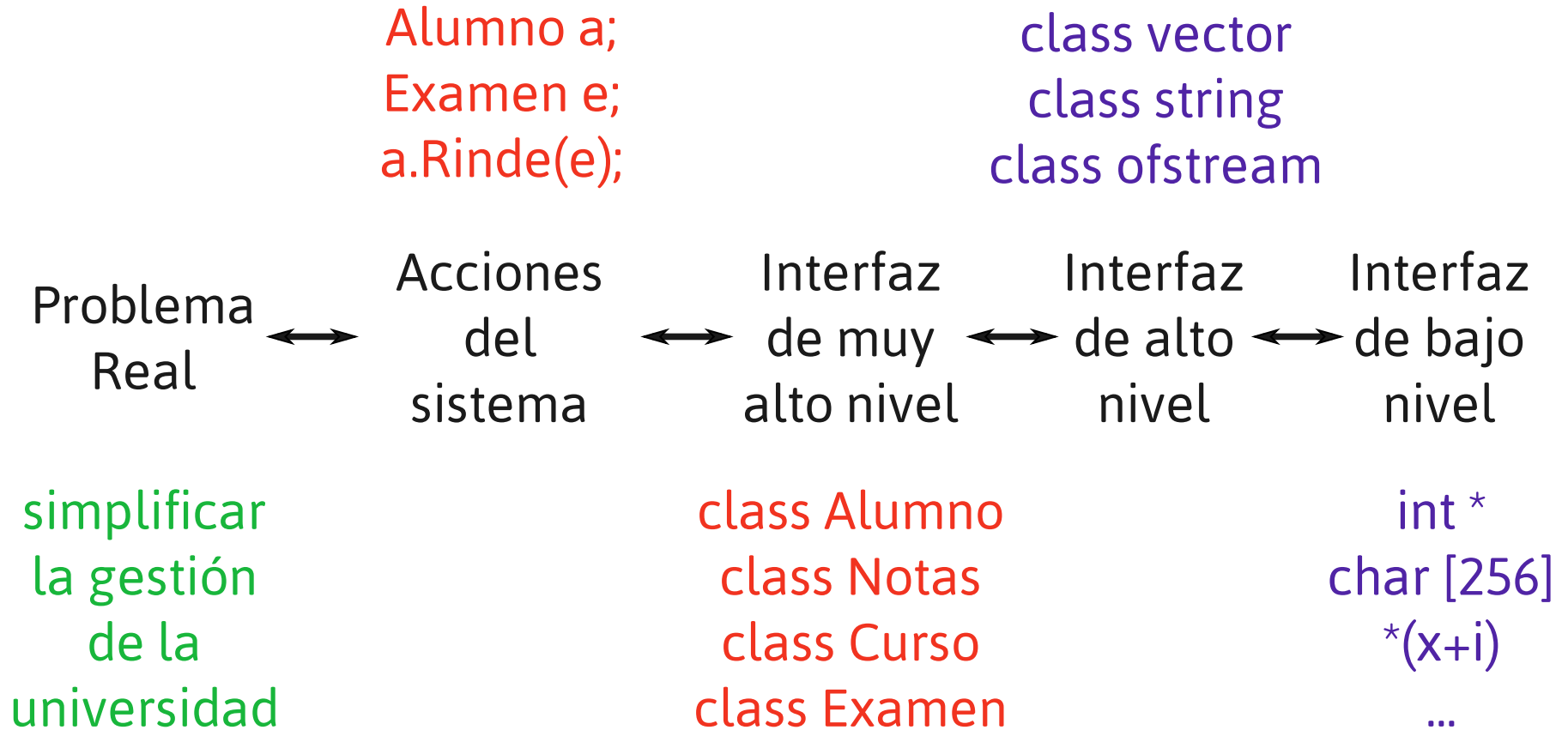
¿Qué mecanismos de abstracción/reducción de la complejidad conocen?

¿QUÉ ES LA POO?

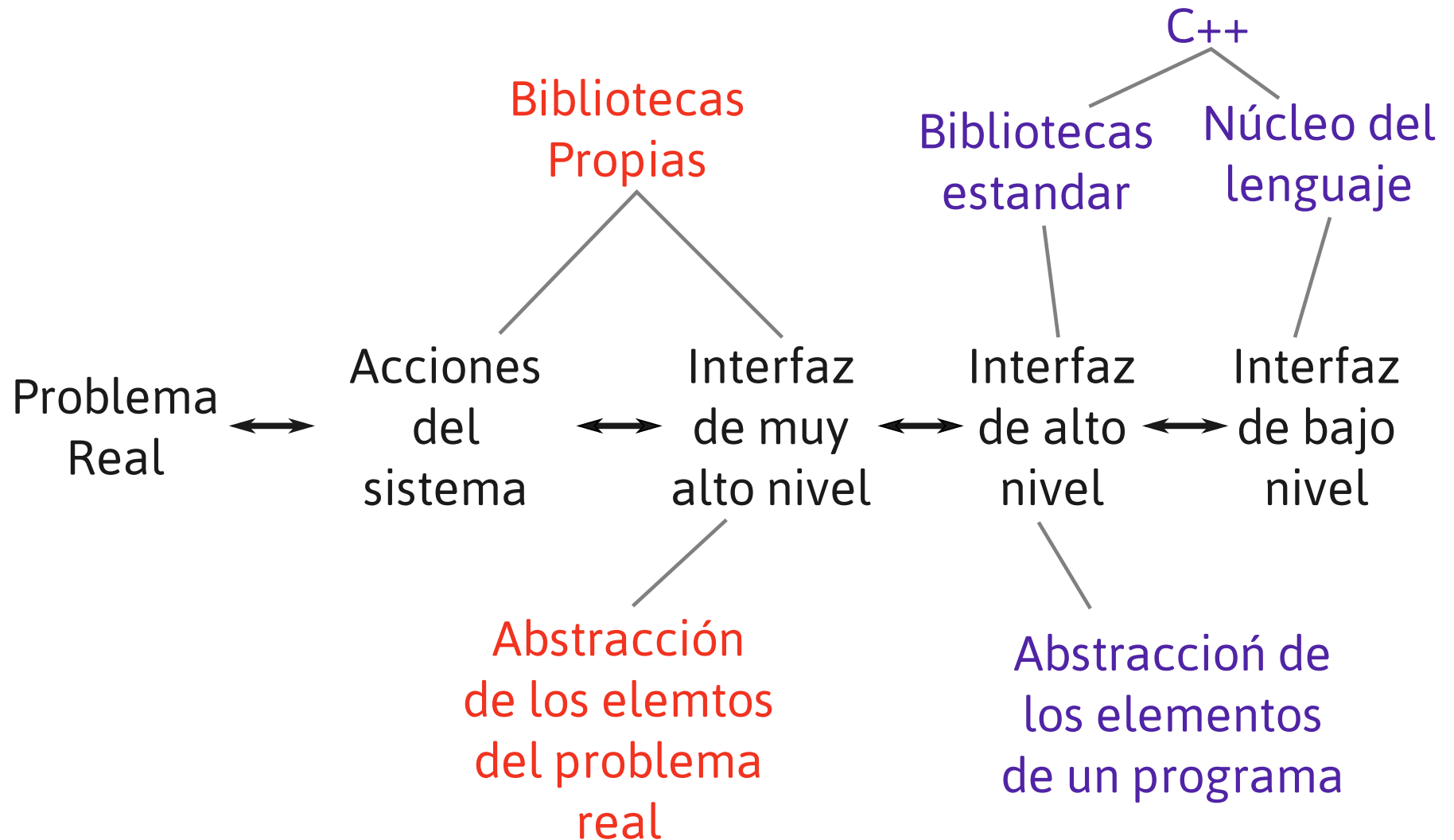
La Programación Orientada a Objetos es un paradigma que utiliza objetos como elementos fundamentales en la construcción de la solución.

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?



¿QUÉ ES LA POO?



¿QUÉ SON LOS OBJETOS?

Objeto: entidad provista de un conjunto de datos o estado, y un conjunto 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> {  
    private:
```

lo que no se ve desde afuera del objeto

```
    public:
```

lo que sí se ve desde afuera del objeto

```
};
```


PRINCIPIO DE OCULTACIÓN

Cada objeto está aislado del exterior, y expone solo una interfaz a 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> {  
    private:  
        <atributos privados>  
        -----  
        estado del objeto  
        <métodos privados>  
        -----  
        funciones auxiliares  
    public:  
        <atributos públicos>  
        -----  
        mala idea  
        <métodos públicos>  
        -----  
        interfaz del objeto  
};
```

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 {  
    // atributos cargados por el usuario  
    float m_a, m_b, m_c;  
    // atributos calculados por el objeto  
    float m_r1, m_r2;  
    bool son_reales;  
    // método auxiliar para resolver los cálculos  
    void Calcular();  
public:  
    // interfaz para carga de datos  
    void CargarCoefs(int a, int b, int c);  
    // interfaz para consulta de resultados  
    bool TieneRaicesReales();  
    float VerRaiz1();  
    float VerRaiz2();  
    float VerParteReal();  
    float VerParteImag();  
};
```

IMPLEMENTACIÓN DE MÉTODOS

Un método implementa como una función global, pero agregando el **scope (nombre de la clase)** al nombre de la función/método:

```
void Ecuacion::CargarCoefs(int a, int b, int c)
{
    // El método CargarCoefs "guarda" los datos que recibe
    // en los atributos del objeto
    m_a = a; m_b = b; m_c = c; Calcular();
}
```

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

IMPLEMENTACIÓN DE MÉTODOS

```
void Ecuacion::Calcular() {  
    float d = m_b*m_b-4*m_a*m_c;  
    if (d>=0) {  
        m_son_reales = true;  
        m_r1 = (-m_b+sqrt(d))/(2*m_a);  
        m_r2 = (-m_b-sqrt(d))/(2*m_a);  
    } else {  
        m_son_reales = false;  
        m_r1 = -m_b/(2*m_a);  
        m_r2 = sqrt(-d)/(2*m_a);  
    }  
}
```

⚠ **El método Calcular no necesita recibir nada**, los datos de entrada ya estarán en los **atributos** gracias al método CargarCoefs

IMPLEMENTACIÓN DE MÉTODOS

```
void Ecuacion::Calcular() {  
    float d = m_b*m_b-4*m_a*m_c;  
    if (d>=0) {  
        m_son_reales = true;  
        m_r1 = (-m_b+sqrt(d))/(2*m_a);  
        m_r2 = (-m_b-sqrt(d))/(2*m_a);  
    } else {  
        m_son_reales = false;  
        m_r1 = -m_b/(2*m_a);  
        m_r2 = sqrt(-d)/(2*m_a);  
    }  
}
```

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

¿CÓMO IDENTIFICAR OBJETOS?

Todos los nombres del enunciado son candidatos:

1. Cosas tangibles (Auto, Arma)
2. Roles o papeles (Alumno, Empleado)
3. Organizaciones (Empresa, Facultad)
4. Incidentes o sucesos (Liquidación, Examen)
5. Interacciones o relaciones (Pedido, Alquiler)

¿CÓMO IDENTIFICAR MÉTODOS Y ATRIBUTOS?

Métodos: verbos

- ▶ Modificadores del estado
- ▶ Cálculos o procesamiento
- ▶ Selectores (consulta)
- ▶ Mezcladores
- ▶ Constructor/Destructor

Atributos: características individuales

- ▶ adjetivos y complementos del verbo en el enunciado
- ▶ lo 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
- 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
- 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 {  
    float m_a, m_b, m_c;  
    float m_r1, m_r2;  
    bool m_son_reales;  
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;  
        m_r1=o.m_r1; m_r2=o.m_r2;  
        m_son_reales=o.m_son_reales;  
    }  
    ~Ecuacion() { /* naaada */ }  
};
```

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*.

EJEMPLO RAI: MEMORIA

```
class Vector {  
    int *m_data;  
public:  
    // adquisición  
    Vector(int n) { m_data = new int[n]; }  
    // liberación  
    ~Vector() { delete [] m_data; }  
};  
  
void foo() {  
    Vector v1(10); // adquisición inevitable  
    ... // uso  
} // liberación automática
```

“ Just that closing brace. Here is where all the magic happens.
Bjarne Stroustrup

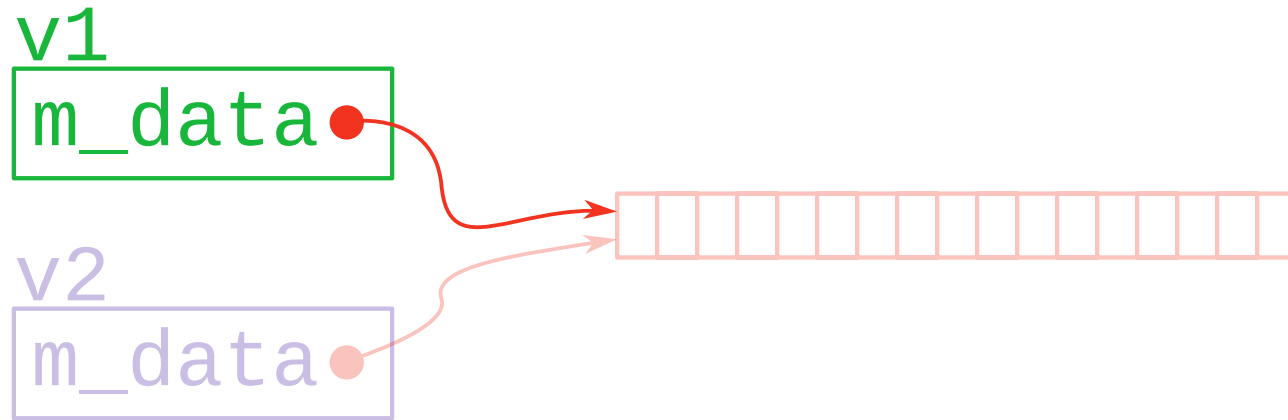
FILOSOFÍA RAI

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
 - El constructor garantiza la adquisición previa al uso
 - El destructor garantiza la liberación luego del uso
- Ejemplos de recursos:
 - archivos, conexiones de red, memoria, ...

CONSTRUCTOR DE COPIA

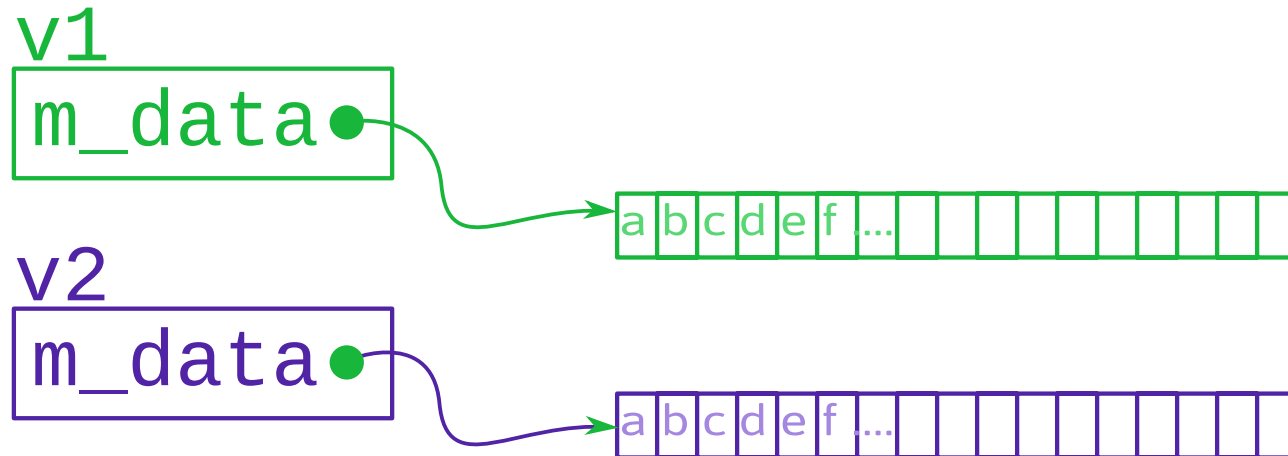
```
int main() {  
    Vector v1(100);  
    ...  
    Vector v2 = v1; // copia, equivale a v2(v1)  
    ...  
} // error: doble delete
```



⚠ 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  
        // 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];  
    }  
}
```

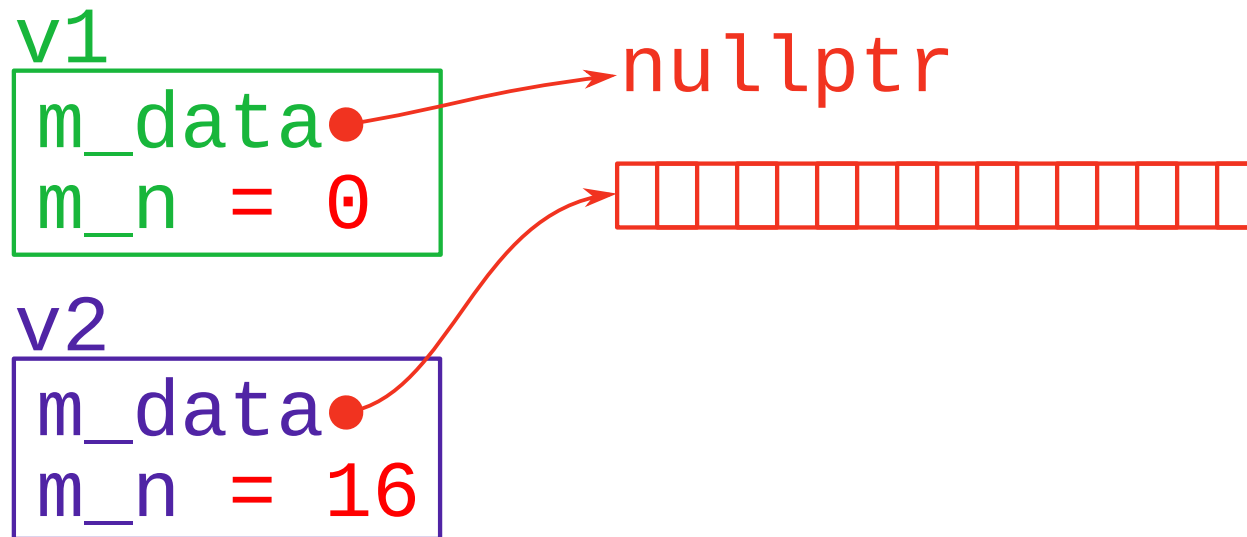
CONSTRUCTOR DE "MOVE"

```
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 {  
    float num;  
public:  
    Calculadora();  
    void Sum (float num);  
    void Rest(float num);  
    void Mult(float num);  
    void Div (float num);  
    float Result();  
};  
  
int main() {  
    Calculadora calc;  
    calc.Sum(9);  
    calc.Rest(3);  
    calc.Mult(7);  
    cout << calc.Result(); // muestra 42  
}
```

EL PUNTERO this

```
class Calculadora {  
    float num;  
public:  
    Calculadora() { this->num = 0.f; }  
    void Sum (float num) { this->num += num; }  
    void Rest(float num) { this->num -= num; }  
    void Mult(float num) { this->num *= num; }  
    void Div (float num) { this->num /= num; }  
    float 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 {  
    float num;  
public:  
    Calculadora() { this->num = 0.f; }  
    Calculadora &Sum(float num) {  
        this->num += num; return *this;  
    }  
    Calculadora &Rest(float num) {  
        this->num -= num; return *this;  
    }  
    Calculadora &Mult(float num) {  
        this->num *= num; return *this;  
    }  
    Calculadora &Div(float num) {  
        this->num /= num; return *this;  
    }  
    float Result() { return this->num; }  
};
```

EL PUNTERO this

```
class Calculadora {  
    float num;  
public:  
    Calculadora() { this->num = 0.f; }  
    Calculadora &Sum(float num) { ... }  
    Calculadora &Rest(float num) { ... }  
    Calculadora &Mult(float num) { ... }  
    Calculadora &Div(float num) { ... }  
    float Result() { return num; }  
};  
  
int main() {  
    Calculadora calc;  
    cout<<calc.Sum(9).Rest(3).Mult(7).Result();  
}
```

MÉTODOS const

```
class Calculadora {  
    public:  
        float Result();  
};
```

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

⊗ No compila!

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


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

MÉTODOS const

```
class Calculadora {
```

```
public:
```

```
    float Result() const;  
};
```

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

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

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

INICIALIZACIÓN DE MIEMBROS

1. Asignar dentro del constructor

```
class Calculadora {  
    float num;  
public:  
    Calculadora() { num = 0.f; }  
    ...  
};
```

INICIALIZACIÓN DE MIEMBROS

2. Asignar en la definición de la variable miembro

```
class Calculadora {  
    float num = 0.f;  
public:  
    Calculadora() { /*nada*/ }  
    ...  
};
```

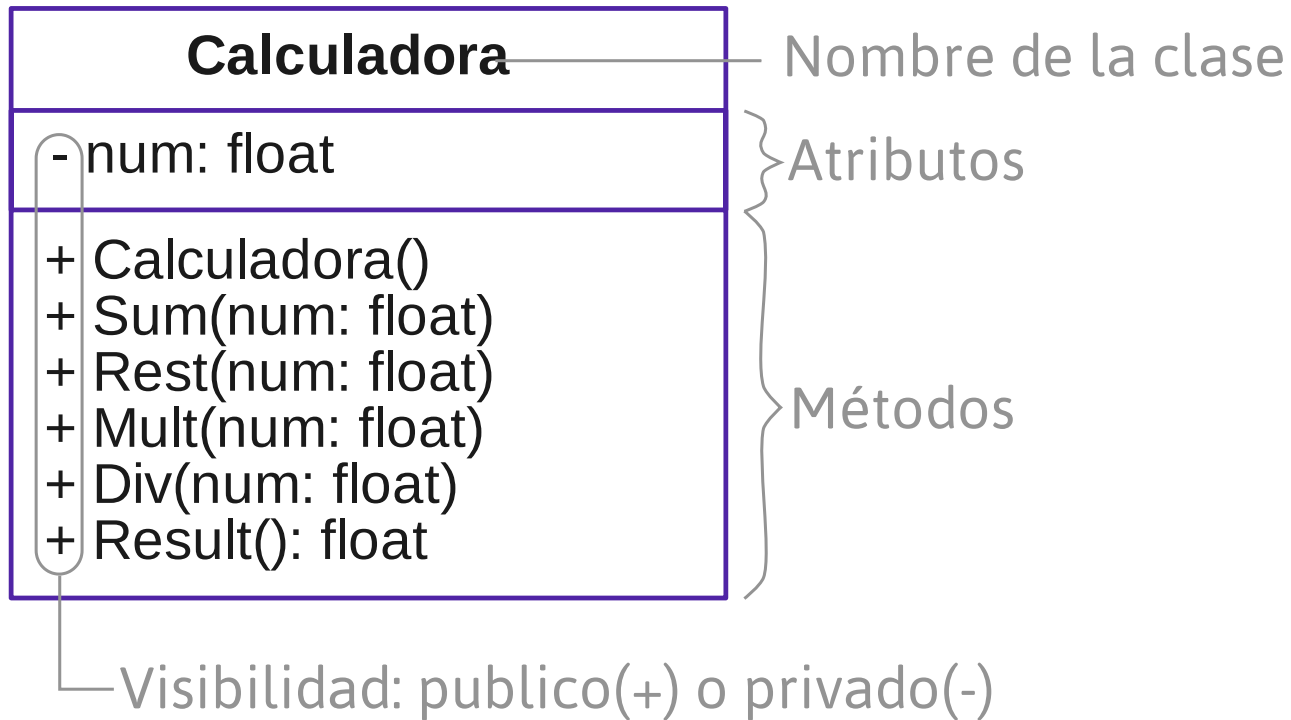
✅ En este caso no haría falta ni declarar el constructor

INICIALIZACIÓN DE MIEMBROS

3. Sintaxis especial para inicialización de variables miembro

```
class Calculadora {  
    float num;  
public:  
    Calculadora() : num(0.f) { /*nada*/ }  
    ...  
};
```

NOTACIÓN UML



ATRIBUTOS Y MÉTODOS `static`

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

f1

| | |
|---|---|
| a | 1 |
| b | 2 |
| c | 3 |

f2

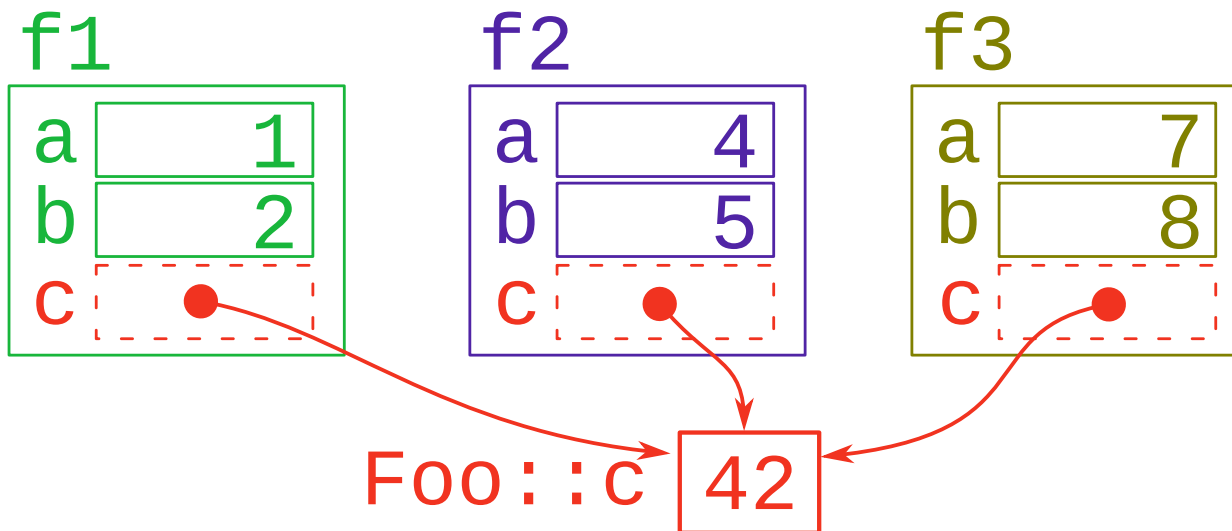
| | |
|---|---|
| a | 4 |
| b | 5 |
| c | 6 |

f3

| | |
|---|---|
| a | 7 |
| b | 8 |
| c | 9 |

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`*