Clases en C++ (Definición)

Algoritmos y Estructuras de Datos II

Definiciones vs. Declaraciones

Declaraciones

Asocian un **nombre** a un **tipo**.

Se pueden repetir.

Ejemplos:

```
class Persona;
bool foo(int x);
extern int x;
```

Definiciones

Son declaraciones que además le otorgan un **valor** al nombre.

No se pueden repetir.

Ejemplos:

```
int x = 5;
int y;
bool foo(int x) {
    return x == 10;
};
```

```
class Persona {
    string nombre;
    string apellido;
};
```

Scopes

- El **scope** es el alcance de una declaración, es decir, el fragmento del programa en el que dicho nombre es visible.
- En C++ los scopes se delimitan por llaves $\{\ldots\}$.
- Los scopes con nombre se pueden acceder mediante ::

```cpp
namespace NS {
    int ns1 = 1;
    int ns2 = ns1 + 1;
}

int global1 = 2;
// int global2 = ns1 + 1;
int global2 = NS::ns2;

int foo() {
    int foo1 = global1 + 1;
    {
        int foo2 = 2;
    }
    // int foo3 = foo2 + 1;  error: 'foo2' was not delcared in this scope
    int x = 4;
    // int x = 5; redeclaration of ‘int x’
    for (int i = 0; i < 5; i++) {
    }
    int i = 10;
}
```

Módulos y Clases: Objetivo

Queremos una caja opaca con palancas que cumpla un cierto comportamiento (una interfaz). Tomar un problema complicado, resolverlo, y empaquetarlo de forma que no sea necesario conocer todos los detalles para usar la solución.

Las clases son una forma de abstraer comportamiento e información.

Vimos ejemplos de esto.

TAD SECUENCIA(α)

parámetros formales

géneros α

observadores básicos

vacía?	:	secu(α)	$\longrightarrow$	bool	
prim	:	secu(α) s	$\longrightarrow$	α	$\{\neg$ vacía?(s)$\}$
fin	:	secu(α) s	$\longrightarrow$	secu(α)	$\{\neg$ vacía?(s)$\}$

generadores

$<>$	:		$\longrightarrow$	secu(α)
$\bullet\bullet\bullet$	:	$\alpha \times$ secu(α)	$\longrightarrow$	secu(α)

Fin TAD

```
template <class t>
class vector<t> {
public:
    // interfaz de la clase
    vector();                   // constructor.
    void push_back(t elem);     // agrega un elemento al final.

    t operator[](int idx);      // devuelve el i-ésimo elemento.
    int size();                 // devuelve el tamaño del vector.
};
```

En TADs, el problema a resolver es *entender* el problema y describirlo en detalle (*especificación*).

En Módulos, el problema a resolver es, una vez entendido el problema, definir una interfaz y su implementación, de forma que puedan usarse las palancas sin entender el detalle de cómo se hace para que la computadora lo pueda ejecutar.

Problema

Modelar una contador de puntos de Truco para dos jugadores.

Necesitamos:

- Conocer el puntaje de ambos jugadores
- Saber si un jugador está en las buenas
- Poder sumar puntos a cada jugador

Este comportamiento podemos declararlo en una interfaz, por ejemplo:

```
class Truco {
    public:
        uint puntaje_j1();
        uint puntaje_j2();

        Truco();
        void sumar_punto(uint);

        bool buenas(uint);
};
```

```
class Truco {
    public:
        uint puntaje_j1();
        uint puntaje_j2();

        Truco();
        void sumar_punto(uint);

        bool buenas(uint);
};
```

Si les diera esta clase en C++, ¿podrían usarla?

```
class Truco {
    public:
        uint puntaje_j1();
        uint puntaje_j2();

        Truco();
        void sumar_punto(uint);

        bool buenas(uint);
};
```

Si les diera esta clase en C++, ¿podrían usarla?

- ¿Cuántos puntos suma `sumar_punto`?
- ¿Qué puntos tiene cada jugador al principio?
- ¿Qué es la información que nos da `buenas`?

TAD TRUCO

observadores básicos

puntajes : Truco $\longrightarrow$ tupla$\langle$Nat, Nat$\rangle$

generadores

NuevaPartida : $\longrightarrow$ Truco

SumarPunto : Truco $\times$ Nat n $\longrightarrow$ Truco $\{n = 1 \vee n = 2\}$

otras operaciones

buenas? : Truco $\longrightarrow$ tupla$\langle$bool, bool$\rangle$

axiomas

puntajes(NuevaPartida) $\equiv$ $\langle 0, 0 \rangle$

puntajes(SumarPunto(t, n)) $\equiv$ puntajes(t) + $\langle$beta(n == 1), beta(n == 2)$\rangle$

buenas?(t) $\equiv$ $\langle$ π_0(puntajes(t)) > 15, π_1(puntajes(t)) $> 15\rangle$

Fin TAD

TAD TRUCO

observadores básicos

puntajes : Truco $\longrightarrow$ tupla$\langle$Nat, Nat$\rangle$

generadores

NuevaPartida : $\longrightarrow$ Truco

SumarPunto : Truco $\times$ Nat n $\longrightarrow$ Truco $\{n = 1 \vee n = 2\}$

otras operaciones

buenas? : Truco $\longrightarrow$ tupla$\langle$bool, bool$\rangle$

axiomas

puntajes(NuevaPartida) $\equiv$ $\langle 0, 0 \rangle$

puntajes(SumarPunto(t, n)) $\equiv$ puntajes(t) + $\langle$beta(n == 1), beta(n == 2)$\rangle$

buenas?(t) $\equiv$ $\langle$ π_0(puntajes(t)) > 15, π_1(puntajes(t)) $> 15\rangle$

Fin TAD

Más adelante vamos a relacionar el TAD con las funciones de C++ escribiendo una interfaz.

Clase e Instancia

Los módulos (Clases) son abstracciones.

```
Truco t1;
t1.sumar_punto(1);
t1.sumar_punto(1);
t1.sumar_punto(1);
t1.sumar_punto(2);
t1.sumar_punto(2);
cout << t1.puntaje_j1(); // 3
cout << t1.puntaje_j2(); // 2
```

```
Truco t2;
t2.sumar_punto(2);
t2.sumar_punto(2);
t2.sumar_punto(1);
t2.sumar_punto(2);
t2.sumar_punto(2);
cout << t2.puntaje1(); // 1
cout << t2.puntaje2(); // 4
```

Clase e Instancia

Los módulos (Clases) son abstracciones.

```
Truco t1;
t1.sumar_punto(1);
t1.sumar_punto(1);
t1.sumar_punto(1);
t1.sumar_punto(2);
t1.sumar_punto(2);
cout << t1.puntaje_j1(); // 3
cout << t1.puntaje_j2(); // 2
```

```
Truco t2;
t2.sumar_punto(2);
t2.sumar_punto(2);
t2.sumar_punto(1);
t2.sumar_punto(2);
t2.sumar_punto(2);
cout << t2.puntaje1(); // 1
cout << t2.puntaje2(); // 4
```

`class Truco` es la abstracción.

`t1` y `t2` son instancias de la abstracción.

Para representar estas diferencias e implementar un código que ejecute el modelo, tenemos que definir una **representación interna**.

La representación es un conjunto de variables que componen la instancia y cuyo estado la definen. Sobre estas variables se ejecutan las operaciones y matenemos información del módulo en el tiempo.

Cómo decidir la representación interna y escribir código que la mantenga correctamente es un **tema principal** de la materia, a desarrollar durante el cuatrimestre. En la práctica se va a ejercitar cómo describir el comportamiento sin tener que definir una representación interna. En el labo vamos a aprender y ejercitar los conceptos técnicos necesarios para poder hacer una representación ejecutable que cumpla los conceptos definidos de forma abstracta en la especificación.

```
class Truco {
  public:
    uint puntaje_j1();
    uint puntaje_j2();

    Truco();
    void sumar_punto(uint);

    bool buenas(uint);

  private:
    uint puntaje_j1_;
    uint puntaje_j2_;
};
```

- Definimos las variables de la representación interna como privadas (no son parte de la interfaz).
- Por costumbre de C++, las variables de la representación interna se terminan con un _.

Estados internos

```
Truco t1;
t1.sumar_punto(1);
t1.sumar_punto(1);
t1.sumar_punto(1);
t1.sumar_punto(2);
t1.sumar_punto(2);
cout << t1.puntaje_j1(); // 3
cout << t1.puntaje_j2(); // 2

// Estado interno t1
t1.puntaje_j1_ == 3;
t1.puntaje_j2_ == 2;
```

```
Truco t2;
t2.sumar_punto(2);
t2.sumar_punto(2);
t2.sumar_punto(1);
t2.sumar_punto(2);
t2.sumar_punto(2);
cout << t2.puntaje_j1(); // 1
cout << t2.puntaje_j2(); // 4

// Estado interno t2
t2.puntaje_j1_ == 1;
t2.puntaje_j2_ == 4;
```

Comportamiento genérico

¿Cómo definimos comportamiento genérico para las instancias?

```
void Truco::sumar_punto(uint j) {
    if (j == 1) {
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++;
    }
};
```

Un *método* es una función asociada a una *clase*.

`this` es el parámetro implícito en métodos. Refiere a la instancia a la que se le está llamando el método.

`this` se accede con el operador `->` en lugar de `.` por ser un *puntero* (a ver en futuras clases).

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1); // <<
    t1.sumar_punto(1);

    t2.sumar_punto(2);
}
```

Contexto

uint	t1.puntaje_j1_	0
uint	t1.puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1); // <<
    t1.sumar_punto(1);

    t2.sumar_punto(2);
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) { // <<
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++;
    }
};
```

Contexto

uint	j	1
uint	this->puntaje_j1_	0
uint	this->puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1); // <<
    t1.sumar_punto(1);

    t2.sumar_punto(2);
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) {
        this->puntaje_j1_++; // <<
    } else {
        this->puntaje_j2_++;
    }
};
```

Contexto

uint	j	1
uint	this->puntaje_j1_	0
uint	this->puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1); // <<
    t1.sumar_punto(1);

    t2.sumar_punto(2);
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) {
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++;
    } // <<
};
```

Contexto

uint	j	1
uint	this->puntaje_j1_	1
uint	this->puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1); // <<

    t2.sumar_punto(2);
}
```

Contexto

uint	t1.puntaje_j1_	1
uint	t1.puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1); // <<

    t2.sumar_punto(2);
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) { // <
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++;
    }
};
```

Contexto

uint	j	1
uint	this->puntaje_j1_	1
uint	this->puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1); // <<

    t2.sumar_punto(2);
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) {
        this->puntaje_j1_++; // <<
    } else {
        this->puntaje_j2_++;
    }
};
```

Contexto

uint	j	1
uint	this->puntaje_j1_	1
uint	this->puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1); // <<

    t2.sumar_punto(2);
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) {
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++;
    }  // <<
};
```

Contexto

uint	j	1
uint	this->puntaje_j1_	2
uint	this->puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1);

    t2.sumar_punto(2); // <<
}
```

Contexto

uint	t1.puntaje_j1_	2
uint	t1.puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1);

    t2.sumar_punto(2); // <<
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) { // <<
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++;
    }
};
```

Contexto

uint	j	2
uint	t1.puntaje_j1_	2
uint	t1.puntaje_j2_	0
uint	this->puntaje_j1_	0
uint	this->puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1);

    t2.sumar_punto(2); // <<
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) {
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++; // <<
    }
};
```

Contexto

uint	j	2
uint	t1.puntaje_j1_	2
uint	t1.puntaje_j2_	0
uint	this->puntaje_j1_	0
uint	this->puntaje_j2_	0

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1);

    t2.sumar_punto(2); // <<
}
```

```
void Truco::sumar_punto(uint j) {
    if (j == 1) {
        this->puntaje_j1_++;
    } else {
        this->puntaje_j2_++;
    } // <<
};
```

Contexto

uint	j	2
uint	t1.puntaje_j1_	2
uint	t1.puntaje_j2_	0
uint	this->puntaje_j1_	0
uint	this->puntaje_j2_	1

Contexto de la instancia

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1);

    t2.sumar_punto(2);
} // <<
```

Contexto

uint	t1.puntaje_j1_	2
uint	t1.puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	1

Contexto de la instancia

El parámetro implícito `this` puede obviarse ya que los miembros internos de una clase aparecen en el contexto de variables accesibles.

```
int main() {
    Truco t1 = Truco();
    Truco t2 = Truco();

    t1.sumar_punto(1);
    t1.sumar_punto(1);

    t2.sumar_punto(2); // <<
}
```

Contexto

uint	t1.puntaje_j1_	2
uint	t1.puntaje_j2_	0
uint	t2.puntaje_j1_	0
uint	t2.puntaje_j2_	1

Contexto de la instancia

```
void Truco::sumar_punto(uint j) {
    if (j == 1) { // <<
        puntaje_j1_++; // this->puntaje_j1_++;
    } else {
        puntaje_j2_++; // this->puntaje_j2_++;
    }
};
```

Contexto

uint	j	2
uint	t1.puntaje_j1_	2
uint	t1.puntaje_j2_	0
uint	puntaje_j1_	0
uint	puntaje_j2_	1

El resto de los ingredientes

La interfaz de Truco tiene métodos para ver el puntaje de los jugadores:

```
class Truco {
  public:
    uint puntaje_j1();
    uint puntaje_j2();
    ...
  private:
    uint puntaje_j1_;
    uint puntaje_j2_;
}
```

Esto se debe a que los miembros privados de una clase no son accessibles por fuera de la misma.

```
int main() {
  Truco t;
  cout << t.puntaje_j1_ << endl;
    // error `uint Truco::puntaje_j1_` is private
}
```

```
uint Truco::puntaje_j1() {
    return this->puntaje_j1_;
}
uint Truco::puntaje_j2() {
    return puntaje_j2_;
}

int main() {
  Truco t;
  t.sumar_punto(1);
  cout << t.puntaje_j1() << endl;  // 1
  cout << t.puntaje_j2() << endl;  // 0
}
```

```
uint Truco::buenas(uint j) {
    if (j == 1) {
        return puntaje_j1_ > 15;
    } else {
        return puntaje_j2_ > 15;
    }
}

int main() {
    Truco t;
    for (uint i = 0; i < 15; i++) {
        t.sumar_punto(1);
        t.sumar_punto(2);
    }
    t.sumar_punto(1);
    cout << t.buenas(1) << endl; // True
    cout << t.buenas(2) << endl; // False
}
```

Constructor

- Los constructores son funciones especiales para inicializar una nueva instancia de un tipo.
- Se escriben con el nombre del tipo.
- No tienen tipo de retorno (está implícito).
- Permiten definir una *lista de inicialización*.

```
Truco::Truco() : puntaje_j1_(0), puntaje_j2_(0) {
}

int main() {
  Truco t = Truco();
  Truco t2;
  Truco t3();
}
```

```
template <class t>
class vector<t> {
public:
    // interfaz de la clase
    vector();                   // constructor.
    void push_back(t elem);     // agrega un elemento al final.

    t operator[](int idx);      // devuelve el i-ésimo elemento.
    int size();                 // devuelve el tamaño del vector.
};
```

```
class Truco {
    public:
        uint puntaje_j1();
        uint puntaje_j2();

        Truco();
        void sumar_punto(uint);

        bool buenas(uint);
};
```

Constructor

- Los constructores son funciones especiales para inicializar una nueva instancia de un tipo.
- Se escriben con el nombre del tipo.
- No tienen tipo de retorno (está implícito).
- Permiten definir una *lista de inicialización*.

```
Truco::Truco() : puntaje_j1_(0), puntaje_j2_(0) {
}

int main() {
  Truco t = Truco();
  Truco t2;
  Truco t3();
}
```

Todo junto

```
#include <ostream>

using namespace std;

class Truco {
  public:
    uint puntaje_j1();
    uint puntaje_j2();

    Truco();
    void sumar_punto(uint);

    bool buenas(uint);

    bool operator==(Truco o);

  private:
    uint puntaje_j1_;
    uint puntaje_j2_;
};

uint Truco::puntaje_j1() {
  return this->puntaje_j1_;
}
```

```
uint Truco::puntaje_j2() {
  return puntaje_j2_;
}

Truco::Truco() : puntaje_j1_(0),
↪  puntaje_j2_(0) {
}

void Truco::sumar_punto(uint j) {
  if (j == 1) {
    puntaje_j1_++;
  } else {
    this->puntaje_j2_++;
  }
}

bool Truco::buenas(uint j) {
  if (j == 1) {
    return puntaje_j1_ > 15;
  } else {
    return puntaje_j2_ > 15;
  }
}
```

Respiramos...

Otro ejercicio: Libreta

Modelar una libreta universitaria. Necesitamos:

- Saber a quién pertenece (LU)
- Saber de qué materias se aprobaron los prácticos
- Saber de qué materias se aprobaron finales
- Conocer la nota del final

Otro ejercicio: Libreta

Modelar una libreta universitaria. Necesitamos:

- Saber a quién pertenece (LU)
- Saber de qué materias se aprobaron los prácticos
- Saber de qué materias se aprobaron finales
- Conocer la nota del final

A responder...

- ¿Interfaz?

Otro ejercicio: Libreta

Modelar una libreta universitaria. Necesitamos:

- Saber a quién pertenece (LU)
- Saber de qué materias se aprobaron los prácticos
- Saber de qué materias se aprobaron finales
- Conocer la nota del final

A responder...

- ¿Interfaz?
  - Observables

Otro ejercicio: Libreta

Modelar una libreta universitaria. Necesitamos:

- Saber a quién pertenece (LU)
- Saber de qué materias se aprobaron los prácticos
- Saber de qué materias se aprobaron finales
- Conocer la nota del final

A responder...

- ¿Interfaz?
  - Observables
  - Modificaciones

Otro ejercicio: Libreta

Modelar una libreta universitaria. Necesitamos:

- Saber a quién pertenece (LU)
- Saber de qué materias se aprobaron los prácticos
- Saber de qué materias se aprobaron finales
- Conocer la nota del final

A responder...

- ¿Interfaz?
  - Observables
  - Modificaciones
  - Constructor

Otro ejercicio: Libreta

Modelar una libreta universitaria. Necesitamos:

- Saber a quién pertenece (LU)
- Saber de qué materias se aprobaron los prácticos
- Saber de qué materias se aprobaron finales
- Conocer la nota del final

A responder...

- ¿Interfaz?
  - Observables
  - Modificaciones
  - Constructor
- ¿Representación?

Libreta

```
using LU = string;
using Materia = string;
using Nota = uint;

class Libreta {
  public:
    LU lu();
    set<Materia> practicos_aprobados();
    set<Materia> finales_aprobados();
    Nota nota_final(Materia m);

    Libreta(LU);
    void aprobar_practico(Materia m);
    void aprobar_final(Materia m, Nota nota);

  private:
    LU lu_;
    set<Materia> practicos_;
    map<Materia, Nota> finales_;
};
```

Algoritmos

```
class Libreta {
  public:
    LU lu();
    set<Materia> practicos_aprobados();
    set<Materia> finales_aprobados();
    Nota nota_final(Materia m);

  private:
    LU lu_;
    set<Materia> practicos_;
    map<Materia, Nota> finales_;
};
```

Algoritmos

```
LU Libreta::lu() {
  return lu_;
}

set<Materia> Libreta::practicos_aprobados() {
  return practicos_;
}

set<Materia> Libreta::finales_aprobados() {
  set<Materia> ret;
  for (pair<Materia, Nota> pn : finales_) {
    ret.insert(pn.first);
  }
  return ret;
}

Nota Libreta::nota_final(Materia m) {
  return finales_.at(m);
}
```

Algoritmos

```
class Libreta {
  public:

    Libreta(LU);
    void aprobar_practico(Materia m);
    void aprobar_final(Materia m, Nota nota);

  private:
    LU lu_;
    set<Materia> practicos_;
    map<Materia, Nota> finales_;
};
```

Algoritmos

```
Libreta::Libreta(LU lu) : lu_(lu), practicos_(), finales_() {}


void Libreta::aprobar_practico(Materia m) {
  practicos_.insert(m);
}

void Libreta::aprobar_final(Materia m, Nota n) {
  practicos_.insert(m);
  finales_.insert(make_pair(m, n));
}
int main() {
  Libreta l("123/04");
  l.aprobar_practico("Algo2");
  l.aprobar_final("Algo1", 10);
  l.practicos_aprobados(); // {Algo1, Algo2}
  l.finales_aprobados(); // {Algo1}
  l.nota_final("Algo1"); // 10
}
```

Lista de inicialización (bis)

¿Qué está pasando acá?

```
class Libreta {
  public:
    ...
  private:
    LU lu_;
    set<Materia> practicos_;
    map<Materia, Nota> finales_;
};
```

```
Libreta::Libreta(LU lu) : lu_(lu), practicos_(), finales_() {}
```

Lista de inicialización (bis)

¿Qué está pasando acá?

```
class Libreta {
  public:
    ...
  private:
    LU lu_;
    set<Materia> practicos_;
    map<Materia, Nota> finales_;
};
```

```
Libreta::Libreta(LU lu) : lu_(lu), practicos_(), finales_() {}
```

Inicializar las variables internas es llamar a constructores de esas variables.

`practicos_()` es llamar al constructor `set<Materia> s();`

Lo último...

Imprimir en pantalla

Cuando queremos ver qué está pasando... imprimimos.

```
int main() {
    Truco t;
    t.sumar_punto(1);
    t.sumar_punto(1);
    t.sumar_punto(1);
    cout << "J1: " << t.puntaje_j1() << " | J2: " <<
    ↪  t.puntaje_j2() << endl;
       // J1: 3 | J2 : 0
}
```

Así como hacemos `<<` de `string` e `int`. ¿No podríamos hacer `cout << t`?

Operadores

Símbolos funcionales como `==`, `+`, `*` y `<<` se conocen como operadores. C++ permite definir funcionamiento para usarlos con nuevos tipos definidos.

Operador <<

Para imprimir se define una función por fuera de la clase de la siguiente manera:

```
ostream& operator<<(ostream& os, Truco t) {
  os << "J1: " << t.puntaje_j1() << " | J2: " << t.puntaje_j2();
  return os;
}
```

Se recibe un `ostream` (`cout` es de ese tipo). Se recibe por referencia (más sobre esto en futuras clases).

Se devuelve el `ostream` para concatenar el operador `<<`

```
Truco t;
cout << t << "(¡Qué partido!)" << endl;
    // J1: 0 | J2: 0 (¡Qué partido!)
```

Comparación por igualdad

```
class Truco {
  public:
    ...
    bool operator==(Truco o);
  private:
    ...
};

bool Truco::operator==(Truco o) {
    return (this->puntaje_j1_ == o.puntaje_j1() and
            this->puntaje_j2() == o.puntaje_j2());
}
```