

Clases en C++ (Definición)

Algoritmos y Estructuras de Datos II

Cualidades del software

Fundamentales:

- ▶ Correcto con respecto a una especificación.

Más o menos importantes, dependiendo del **contexto de uso**:

- ▶ Eficiente (tiempo, memoria, consumo de energía, ...).
- ▶ Reutilizable.
- ▶ Extensible / modificable.
- ▶ Usable.
- ▶ Legible.
- ▶ Predecible.
- ▶ ...

El **diseño** consiste en organizar el programa de tal manera que cumpla con las cualidades requeridas, en algún contexto de uso.

Receta para el desastre

Entender un tipo a través de su **estructura**.

Ejemplo

“Un diccionario es una secu<tupla<clave, valor>>.”



Si queremos modificar el diccionario para saber a qué hora se insertó cada clave por última vez:

- ▶ Hay que cambiar la estructura a:
 secu<tupla<clave, tupla<valor, hora>>>.
- ▶ Hay que cambiar las inserciones para que agreguen la hora.
- ▶ Hay que cambiar todas las búsquedas para quedarse sólo con el valor.
- ▶ Estos cambios pueden estar desperdigados por el programa.

Diseño por contratos

Entender un tipo a través de su **interfaz**.

Ejemplo

“Un diccionario provee operaciones:



1. Crear un diccionario vacío.
2. Asociar una clave a un valor.
3. Buscar una clave.”

Si queremos modificar el diccionario para saber a qué hora se insertó cada clave por última vez:

- ▶ Basta con modificar la implementación (privada) del diccionario.
- ▶ La interfaz se extiende, pero las operaciones anteriores siguen funcionando.

Módulos y clases: objetivo

Un *módulo* es una entidad que tiene:

- ▶ Una interfaz pública con operaciones accesibles para el usuario.
- ▶ Una estructura de representación con mecanismos concretos de implementación de las operaciones, inaccesibles para el usuario.

Las *clases* son la herramienta que vamos a usar para conseguir este comportamiento en C++.

Módulos y clases: objetivo

Un *módulo* es una entidad que tiene:

- ▶ Una interfaz pública con operaciones accesibles para el usuario.
- ▶ Una estructura de representación con mecanismos concretos de implementación de las operaciones, inaccesibles para el usuario.

Las *clases* son la herramienta que vamos a usar para conseguir este comportamiento en C++.

- ▶ Se puede programar modularmente sin clases.
- ▶ Se puede programar con clases pero no modularmente.

Declaraciones vs. definiciones

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;  
};
```

Scope

- ▶ El “**scope**” es el *alcance* de la declaración de un nombre: el fragmento del programa en el que el nombre es visible.
- ▶ En C++ los *scopes* se delimitan por llaves `{ ... }`.
- ▶ Se puede hacer referencia al nombre `x` dentro de un *scope* `n` con la sintaxis `n:x`.


```

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 declared in this scope
    int x = 4;
    // int x = 5; redeclaration of 'int x'
    for (int i = 0; i < 5; i++) {
    }
    int i = 10;
}

```

Ejemplo

Modelar un contador de puntos del juego 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 expresarlo a través de una interfaz.

```
typedef unsigned int uint;

class Truco {
public:
    Truco();
    void sumar_punto(uint);
    uint puntaje(uint);
    bool buenas(uint);
};
```

```
typedef unsigned int uint;

class Truco {
public:
    Truco();
    void sumar_punto(uint);
    uint puntaje(uint);
    bool buenas(uint);
};
```

Si tuviéramos esta clase en C++, ¿sabríamos cómo usarla?

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

TAD TRUCO

observadores básicos

tantos : truco \times nat $j \longrightarrow$ nat $\{j = 1 \vee j = 2\}$

generadores

nuevaPartida : \longrightarrow truco

sumarPunto : truco \times nat $j \longrightarrow$ truco $\{j = 1 \vee j = 2\}$

otras operaciones

puntaje : truco \times nat $j \longrightarrow$ nat $\{j = 1 \vee j = 2\}$

buenas? : truco \times nat $j \longrightarrow$ bool $\{j = 1 \vee j = 2\}$

axiomas

tantos(nuevaPartida(), j) \equiv 0

tantos(sumarPunto(t, j), j') \equiv if $j = j'$ then 1 else 0 fi
+ tantos(t, j')

buenas?(t, j) \equiv tantos(t, j) > 15

puntaje(t, j) \equiv if buenas?(t, j) then
 tantos(t, j) - 15
else
 tantos(t, j)
fi

Clases e instancias

```
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(1); // 3  
cout << t1.puntaje(2); // 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(1); // 1  
cout << t2.puntaje(2); // 4
```

Clases e instancias

```
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(1); // 3  
cout << t1.puntaje(2); // 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(1); // 1  
cout << t2.puntaje(2); // 4
```

`class Truco` es la abstracción.

t1 y t2 son instancias de la abstracción.

Para que el comportamiento de la clase pueda llevarse a cabo, hay que implementarla.

La **implementación** está dada por:

- ▶ La **representación** interna: un conjunto de variables que determinan el estado interno de la instancia.
- ▶ Un conjunto de **algoritmos** que implementan cada una de las operaciones de la interfaz, consultando y modificando las variables de la representación interna.


```
class Truco {  
    public:  
        Truco();  
        void sumar_punto(uint);  
        uint puntaje(uint);  
        bool buenas(uint);  
  
    private:  
        uint _puntaje1;  
        uint _puntaje2;  
        bool _buenas1;  
        bool _buenas2;  
};
```

- Definimos las variables de la representación interna como privadas (no son parte de la interfaz).

Estados internos

```
Truco t1;  
t1.sumar_punto(1);  
t1.sumar_punto(1);  
t1.sumar_punto(1);  
t1.sumar_punto(2);  
t1.sumar_punto(2);
```

```
// Estado interno t1  
t1._puntaje1 == 3;  
t1._puntaje2 == 2;  
t1._buenas1  == false;  
t1._buenas2  == false;
```

```
Truco t2;  
t2.sumar_punto(2);  
t2.sumar_punto(2);  
t2.sumar_punto(1);  
t2.sumar_punto(2);  
t2.sumar_punto(2);
```

```
// Estado interno t2  
t2._puntaje1 == 1;  
t2._puntaje2 == 4;  
t2._buenas1  == false;  
t2._buenas2  == false;
```

Comportamiento genérico

¿Cómo definimos comportamiento genérico para las instancias?

Comportamiento genérico

¿Cómo definimos comportamiento genérico para las instancias?

A través de **métodos**:

```
void Truco::sumar_punto(uint jugador) {  
    if (jugador == 1) {  
        _puntaje1++;  
        if (_puntaje1 == 16) {  
            _puntaje1 = 0;  
            _buenas1 = true;  
        }  
    } else {  
        _puntaje2++;  
        if (_puntaje2 == 16) {  
            _puntaje2 = 0;  
            _buenas2 = true;  
        }  
    }  
}
```

Ejemplo

```
int main() {  
    Truco t1 = Truco();  
    Truco t2 = Truco();  
                                     // <---  
    t1.sumar_punto(1);  
    t1.sumar_punto(1);  
    t2.sumar_punto(2);  
}
```

Contexto

| | |
|--------------|-------|
| t1._puntaje1 | 0 |
| t1._puntaje2 | 0 |
| t1._buenas1 | false |
| t1._buenas2 | false |
| t2._puntaje1 | 0 |
| t2._puntaje2 | 0 |
| t2._buenas1 | false |
| t2._buenas2 | false |

Ejemplo

```
int main() {  
    Truco t1 = Truco();  
    Truco t2 = Truco();  
    t1.sumar_punto(1);  
                                     // <---  
    t1.sumar_punto(1);  
    t2.sumar_punto(2);  
}
```

Contexto

| | |
|--------------|-------|
| t1._puntaje1 | 1 |
| t1._puntaje2 | 0 |
| t1._buenas1 | false |
| t1._buenas2 | false |
| t2._puntaje1 | 0 |
| t2._puntaje2 | 0 |
| t2._buenas1 | false |
| t2._buenas2 | false |

Ejemplo

```
int main() {  
    Truco t1 = Truco();  
    Truco t2 = Truco();  
    t1.sumar_punto(1);  
    t1.sumar_punto(1);  
                                     // <---  
    t2.sumar_punto(2);  
}
```

Contexto

| | |
|--------------|-------|
| t1._puntaje1 | 2 |
| t1._puntaje2 | 0 |
| t1._buenas1 | false |
| t1._buenas2 | false |
| t2._puntaje1 | 0 |
| t2._puntaje2 | 0 |
| t2._buenas1 | false |
| t2._buenas2 | false |

Ejemplo

```
int main() {  
    Truco t1 = Truco();  
    Truco t2 = Truco();  
    t1.sumar_punto(1);  
    t1.sumar_punto(1);  
    t2.sumar_punto(2);  
                                     // <---  
}
```

Contexto

| | |
|--------------|-------|
| t1._puntaje1 | 2 |
| t1._puntaje2 | 0 |
| t1._buenas1 | false |
| t1._buenas2 | false |
| t2._puntaje1 | 0 |
| t2._puntaje2 | 1 |
| t2._buenas1 | false |
| t2._buenas2 | false |

El resto de los ingredientes

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

```
class Truco {  
    public:  
        uint puntaje(uint jugador);  
        ...  
    private:  
        uint _puntaje1;  
        uint _puntaje2;  
        bool _buenas1;  
        bool _buenas2;  
}
```

Pero los miembros privados de una clase no son accesibles desde afuera:

```
int main() {  
    Truco t;  
    cout << t._puntaje1 << endl;  
    // error `uint Truco::_puntaje1` is private  
}
```

```
uint Truco::puntaje(uint jugador) {  
    if (jugador == 1) {  
        return _puntaje1;  
    } else {  
        return _puntaje2;  
    }  
}  
  
int main() {  
    Truco t;  
    t.sumar_punto(1);  
    cout << t.puntaje(1) << endl;    // 1  
    cout << t.puntaje(2) << endl;    // 0  
}
```

```
uint Truco::buenas(uint jugador) {  
    if (jugador == 1) {  
        return _buenas1;  
    } else {  
        return _buenas2;  
    }  
}  
  
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() : _puntaje1_(0), _puntaje2(0),  
                _buenas1(false), _buenas2(false) {  
}
```

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

```
class Truco {  
    public:  
        Truco();  
        void sumar_punto(uint);  
        uint puntaje(uint);  
        bool buenas(uint);  
};
```

Todo junto

```
#include <ostream>
using namespace std;
typedef unsigned int uint;

class Truco {
public:
    Truco();
    void sumar_punto(uint);
    uint puntaje(uint jugador);
    bool buenas(uint jugador);
private:
    uint _puntaje1;
    uint _puntaje2;
    bool _buenas1;
    bool _buenas2;
};

Truco::Truco() :
    _puntaje1(0), _puntaje2(0), _buenas1(false),
    ↪ _buenas2(false) {
}

uint Truco::puntaje(uint jugador) {
    if (jugador == 1) {
        return _puntaje1;
    } else {
        return _puntaje2;
    }
}
```

```
void Truco::sumar_punto(uint jugador) {
    if (jugador == 1) {
        _puntaje1++;
        if (_puntaje1 == 16) {
            _puntaje1 = 0;
            _buenas1 = true;
        }
    } else {
        _puntaje2++;
        if (_puntaje2 == 16) {
            _puntaje2 = 0;
            _buenas2 = true;
        }
    }
}

bool Truco::buenas(uint jugador) {
    if (jugador == 1) {
        return _buenas1;
    } else {
        return _buenas2;
    }
}
```

Otro ejemplo: libreta

Diseñar una libreta universitaria. Queremos:

- ▶ Saber a quién pertenece (número de libreta).
- ▶ Saber de qué materias se aprobaron los prácticos.
- ▶ Saber de qué materias se aprobaron finales.
- ▶ Conocer las notas de los finales.

Otro ejemplo: libreta

Diseñar una libreta universitaria. Queremos:

- ▶ Saber a quién pertenece (número de libreta).
- ▶ Saber de qué materias se aprobaron los prácticos.
- ▶ Saber de qué materias se aprobaron finales.
- ▶ Conocer las notas de los finales.

Necesitamos proponer:

- ▶ Interfaz.
- ▶ Representación.

Libreta

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

class Libreta {
public:
    Libreta(LU);
    LU numero();
    set<Materia> practicos_aprobados();
    set<Materia> finales_aprobados();
    Nota nota_final(Materia m);
    void aprobar_practico(Materia m);
    void aprobar_final(Materia m, Nota nota);

private:
    LU _numero;
    set<Materia> _practicos;
    map<Materia, Nota> _finales;
};
```

Algoritmos

```
Libreta::Libreta(LU lu) :  
    _numero(lu), _practicos(), _finales() {  
}  
  
LU Libreta::numero() {  
    return _numero;  
}  
  
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;  
}
```

Algoritmos

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

Algoritmos

```
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

¿Qué está pasando acá?

```
class Libreta {  
    public:  
        ...  
    private:  
        LU _lu;  
        set<Materia> _practicos;  
        map<Materia, Nota> _finales;  
};  
Libreta::Libreta(LU lu) :  
    _numero(lu), _practicos(), _finales() {  
}
```

Lista de inicialización

¿Qué está pasando acá?

```
class Libreta {  
    public:  
        ...  
    private:  
        LU _lu;  
        set<Materia> _practicos;  
        map<Materia, Nota> _finales;  
};  
  
Libreta::Libreta(LU lu) :  
    _numero(lu), _practicos(), _finales() {  
}
```

Las inicializaciones de las variables son invocaciones a constructores.

`_practicos()` invoca al constructor `set<Materia>()`;

Imprimir en pantalla

```
int main() {  
    Truco t;  
    t.sumar_punto(1);  
    t.sumar_punto(1);  
    t.sumar_punto(1);  
    cout << "J1: " << t.puntaje(1) << endl;  
    cout << "J2: " << t.puntaje(2) << endl;  
}
```

Así como hacemos `cout << x` cuando `x` es un `string` o un `int`, ¿podríamos hacer `cout << t` cuando `t` es un `Truco`?

Operadores

C++ permite extender los operadores como `==`, `+`, `*`, `<<` para que funcionen con tipos definidos por el programador.

Operador "<<"

Para imprimir se define una función por fuera de la clase; por ejemplo:

```
ostream& operator<<(ostream& os, Truco t) {  
    os << "J1: " << t.puntaje(1) << endl;  
    os << "J2: " << t.puntaje(2) << endl;  
    return os;  
}
```

Operador “==”

```
class Truco {  
    public:  
        ...  
        bool operator==(Truco otro);  
    private:  
        ...  
};  
  
bool Truco::operator==(Truco otro) {  
    return _puntaje1 == otro._puntaje1  
        && _puntaje2 == otro._puntaje2  
        && _buenas1 == otro._buenas1  
        && _buenas2 == otro._buenas2;  
}
```